

高等学校计算机系列规划教材

Java EE Web 编程 技术教程

刘甫迎 饶 斌 郑显举 杨雅志 编著

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书详细介绍了 Java EE 基础；Java EE 的可视化集成开发平台（Eclipse 及运行环境）、Java Applet 及 JDBC、Web 层编程技术、Java EE 轻型框架技术、EJB 技术、Java EE 持久性数据管理、Web 服务与 SOA 技术、Java 消息服务等异步技术和 Java EE 综合使用实例。本书既突出轻型框架 Hibernate、Struts 2、Spring，又有企业 JavaBean（EJB 3.0）分布式重型框架；既有 Web 服务，也有面向服务结构（SOA）新技术，其内容主要集中在企业级 Java 项目所需的重要的 API 和工具上，使本书成为一本较完整的 Java EE Web 编程技术教程。本书共 10 章及 3 个附录，有实例、习题、教学大纲和实验指导书等。

本书可作为本科院校、高职院校计算机及相关专业课程教材，也适合 Web 编程开发人员使用、参考。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

Java EE Web 编程技术教程 / 刘甫迎，饶斌，郑显举编著. —北京：电子工业出版社，2010.7

（高等学校计算机系列规划教材）

ISBN 978-7-121-06504-0

I. ①J… II. ①刘… ②饶… ③郑… III. ①JAVA 语言—程序设计—高等学校—教材 IV. ①TP312

中国版本图书馆 CIP 数据核字（2010）第 118611 号

策划编辑：吕 迈

责任编辑：毕军志

印 刷：

北京市李史山胶印厂

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1092 1/16 印张：24 字数：614.4 千字

印 次：2010 年 7 月第 1 次印刷

印 数：4 000 册 定价：39.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zltts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前言

在当今网络时代，无论因特网（Internet）、内联网（Intranet）、外联网（Extranet）都离不开 Web 技术的应用，其使用规模和水平已成为衡量某个国家信息化程度的一个重要标志。Web 编程技术已成为计算机领域中最重要技术之一，它是软件学科中一个不可或缺的分支，是高等学校计算机专业和信息管理专业一门专业基础课，越来越多的人希望学习 Web 编程技术。

随着网络技术尤其是 Web 应用技术的发展，企业级应用对系统各方面的性能要求越来越高，特别是速度、安全、可靠性以及分布式应用等方面，在一定程度上决定着系统能否成功。在这些要求的共同作用下，SUN 的 Java EE（Java 平台企业版）规范利用 Java 编程语言和企业 API 的强大功能，包括 EJB 技术，提供了一种业界领先的 Web 编程技术平台。较之微软的 .NET 平台，Java EE 更加适宜大型企业级项目的开发（对中小型企业项目更是游刃有余）。

企业级 Web 应用技术呼唤一本较完整、实用的《Java EE Web 编程技术教程》出台，本教科书拟适应此需要。

本书的主要特点：

（1）体现最新技术。Java EE（Java 平台企业版）是 J2EE 1.5 以后的称谓，是替代日益成熟的 J2EE 的革命性规范。全书以 Java EE 为基础，体现了内容的先进性（详见第 1 章）。

（2）突出主流技术。例如，叙述了 Java EE 的可视化集成开发平台主流技术 Eclipse、其运行环境的 Web 服务器主流技术 Tomcat 和应用服务器主流技术 JBoss，框架环境采用 MyEclipse，并且将之贯穿全书（见第 2 章、第 5 章）。

（3）注重基础性内容。例如，叙述了 Java EE 规范中的 Java Applet（小程序）及 JDBC，为后面章节学习提供了基础（见第 3 章）。

（4）注意全面性。既突出 Java EE 轻型框架 Struts、Hibernate、Spring 的重点（见第 5 章），又有企业 JavaBean（EJB 3.0）分布式可复用组件重型框架的内容（见第 6 章）；既有 Java EE 的 Web 服务，也有面向服务结构（SOA）新技术（见第 8 章）。还叙述了 JMS、Ajax 异步技术（见第 9 章）。

（5）将 JSP、Servlet、JSTL、JSF 作为 Web 层编程叙述（见第 4 章），并把 Java EE 持久性数据管理内容单独作为一章（见第 7 章）。

（6）本书注重理论与实践相结合，突出实践动手能力和实用性。有实例（见第 10 章）、实验指导书（见附录 B），便于读者参考、使用，力图使学生学习本书后便基本可以编制基于 Java EE 的 Web 应用系统。

（7）本书附有教学大纲（见附录 A）、习题，图文并茂，便于学习与教学。

（8）本书作者长期从事 Web 编程技术教材的编写、教学和科研工作，有丰富的教学和

开发经验，并将其融入本书中。

本书由刘甫迎、饶斌、郑显举、杨雅志编著。刘甫迎编著第 1 章、第 3 章、第 4 章、第 9 章；饶斌编著第 2 章、第 6 章、第 8 章、第 10 章；郑显举编著第 7 章和附录 A、B、C；杨雅志编著第 5 章；全书由刘甫迎统稿。在编著过程中谢林芮、曾克蓉、李朝蓉等做了许多辅助工作，在此一并表示感谢！

由于水平有限，错误难免，请斧正。

刘甫迎

2010 年 4 月

目 录

第 1 章	Java EE 基础	1
1.1	Web 应用基本概念	1
1.1.1	Web 应用定义	1
1.1.2	Web 应用体系结构	2
1.1.3	基于层的设计	6
1.2	Java EE 规范	9
1.2.1	什么是 Java EE	9
1.2.2	Java EE 的体系结构	9
1.2.3	Java EE 应用程序构成及应用	10
1.2.4	几个典型 Java EE 体系结构	14
1.3	Java EE Web 应用的编译和部署	16
1.3.1	Java EE 的部署问题	16
1.3.2	创建一个 JSP 应用程序的实例	19
1.4	Java EE 的发展与特点	21
1.4.1	Java EE 的由来与发展	21
1.4.2	Java EE 的新功能	22
1.4.3	Java EE 开发环境 IDE	25
习题 1		26
第 2 章	Java EE 的可视化集成开发平台——Eclipse 及运行环境	27
2.1	Eclipse 概述	27
2.1.1	Eclipse 的主要特点	27
2.1.2	Eclipse 的组成	28
2.2	Eclipse 的安装及开发环境的搭建	30
2.2.1	下载和安装 JDK	30
2.2.2	下载并解压缩 Eclipse SDK	31
2.2.3	安装 Eclipse 插件	35
2.3	Eclipse 插件的开发及分类	36
2.3.1	基于插件的体系结构	36
2.3.2	开发 HelloWorldPlugin 插件	37
2.3.3	Eclipse 插件的分类	40
2.4	Web 服务器和应用服务器	41
2.4.1	Web 服务器和应用服务器简介	41
2.4.2	Tomcat Web 服务器	43
2.4.3	Eclipse 与 Tomcat 集成	52
2.4.4	JBoss 应用服务器	53

2.4.5 Eclipse 与 JBoss 集成——JBossIDE	57
习题 2	59
第 3 章 Java Applet 及 JDBC	61
3.1 Java Applet 基础	61
3.1.1 在 HTML 中调用 Applet	61
3.1.2 编写一个 Applet	62
3.1.3 改变标签的字体	64
3.1.4 向 Applet 添加文本框和按钮组件	65
3.1.5 Applet 的事件驱动编程	66
3.1.6 添加输出到一个 Applet	69
3.2 Applet 的生命周期和更复杂的 Applet	70
3.2.1 Applet 的生命周期	70
3.2.2 一个全交互的 Applet	73
3.2.3 使用 setLocation()方法	76
3.2.4 使用 setEnable()方法	77
3.2.5 得到帮助	77
3.3 JDBC 及其应用	78
3.3.1 JDBC 编程技术	78
3.3.2 使用 JDBC 访问数据库	80
3.3.3 应用实例	85
习题 3	89
第 4 章 Web 层编程技术	93
4.1 JSP 技术	93
4.1.1 JSP 简介	93
4.1.2 JSP 的语法	95
4.1.3 JSP 的内建对象	98
4.1.4 JSP 的表单及 Cookie 应用	101
4.1.5 JSP 与 JavaBean	104
4.2 Java Servlet 技术	108
4.2.1 Servlet 概述	108
4.2.2 开发 Servlet 应用	116
4.2.3 Servlet 与 JSP、JavaBean 协同工作	117
4.3 用 JSP 访问数据库	119
4.3.1 用 JSP 访问 SQL Server 数据库	119
4.3.2 JSP 用 JavaBean 操纵数据库	120
4.4 JSTL 标准标签库技术	121
4.4.1 JSTL 及其操作实现	121
4.4.2 在 JSP 中使用 JSTL	125
4.5 JSF 技术	129
4.5.1 JSF 及其安装	129

4.5.2	JSP 页面中使用 JSF	131
习题 4		138
第 5 章	Java EE 轻型框架技术	139
5.1	Java EE 轻型框架技术概述	139
5.1.1	轻型框架的流行	139
5.1.2	流行的轻型框架组合	140
5.1.3	轻型框架的 MyEclipse 环境	140
5.2	Struts2 框架	141
5.2.1	Struts 框架及其 MVC 结构	141
5.2.2	Struts2 与 WebWork 在代码重用性上的优势	142
5.2.3	Struts2 的引例、Filter 及配置	147
5.2.4	Struts2 的 Action	156
5.2.5	Struts2 的 OGNL 表达式	160
5.2.6	Struts2 的标签库	162
5.3	Hibernate 框架	164
5.3.1	Hibernate 概述	165
5.3.2	Hibernate 的运行及其映射、基本配置和接口	166
5.3.3	DAO 模式、Hibernate Synchronizer 插件及开发	175
5.3.4	Criteria Query、HQL 数据查询语言及 Query 接口	184
5.3.5	Hibernate 的数据关联	191
5.3.6	Hibernate 实体对象生命周期、缓存管理、事务	198
5.3.7	在 Web 环境下使用 Hibernate	203
5.4	Spring 框架	208
5.4.1	Spring 基础及其开发环境	208
5.4.2	Spring 的 IoC、容器及基本配置	212
5.4.3	Spring 的 AOP	219
5.4.4	Spring 整合 Hibernate	229
5.5	开发 Struts2、Hibernate、Spring 集成程序	237
习题 5		245
第 6 章	EJB 技术	246
6.1	企业级 JavaBean (EJB): Java EE 解决方案及其特点	246
6.2	EJB 的工作原理、环境及运行	247
6.2.1	EJB 的工作原理及类型	247
6.2.2	EJB 3.0 的特点及运行实例	249
6.2.3	独立的 Tomcat 调用 EJB	254
6.2.4	EJB 的类和接口	254
6.3	会话 Bean	255
6.3.1	无状态会话 Bean	256
6.3.2	有状态会话 Bean	256
6.4	消息驱动 Bean	258

6.5 实体 Bean	260
6.5.1 实体 Bean 配置文件及 JBoss 的数据源	261
6.5.2 单表实体 Bean 及持久化实体管理器	262
习题 6	269
第 7 章 Java EE 持久性数据管理	270
7.1 Java 持久性 API 简介	270
7.1.1 实体	270
7.1.2 管理实体	277
7.2 Web 层持久性	281
7.2.1 定义持久性单元	282
7.2.2 创建一个实体类	282
7.2.3 获取对一个实体管理器的访问	283
7.2.4 访问数据库中的数据	285
7.2.5 更新数据库中的数据	285
7.3 EJB 层的持久性（多表实体 Bean）	287
习题 7	291
第 8 章 Web 服务与 SOA 技术	292
8.1 Web 服务到底是什么	292
8.2 Web 服务技术	294
8.2.1 概述	294
8.2.2 XML：自描述数据（DTD 和模式语言、解析 XML）	296
8.3 用 JAX-WS 开发 Web 服务	302
8.3.1 简介 JAX-WS	302
8.3.2 下载 CVS 工具	303
8.3.3 创建 Web 服务	304
8.3.4 构建、测试和运行 Web 服务	309
8.4 面向服务结构	310
8.4.1 SOA 简介	310
8.4.2 SOA 的基础架构	313
8.4.3 SOA 的实现	315
8.4.4 SOA 的未来	320
习题 8	321
第 9 章 Java 消息服务等异步技术	322
9.1 Ajax 技术	322
9.1.1 Asynchronous JavaScript+XML	322
9.1.2 XMLHttpRequest	323
9.1.3 基于 Ajax 的用户注册实例	325
9.1.4 Ajax 集成技术：DWR	325
9.2 Java 消息服务概念	326
9.2.1 什么是 Java 消息服务	326

9.2.2 提供者、客户、消息与管理对象	328
9.3 JMS 编程模型	330
9.3.1 两种 JMS 编程模型	330
9.3.2 特定于模型的管理对象接口	331
9.3.3 消息使用的异步性	331
9.4 JMS 可靠性与性能	332
9.4.1 客户确认	332
9.4.2 消息持久保存	332
9.4.3 时间依赖性和 JMS 发布模型	333
9.5 一个 JMS pub/sub 应用实例	333
9.5.1 开发消息发布者	334
9.5.2 开发消息预约者	335
9.5.3 关于部署	338
习题 9	338
第 10 章 Java EE 综合应用实例——公文管理信息系统	339
10.1 公文管理信息系统概述	339
10.2 设计数据库	340
10.3 系统公共配置	341
10.3.1 导入相关类库	341
10.3.2 配置 web.xml	341
10.3.3 数据源配置	342
10.3.4 配置 persistence.xml 文件	342
10.4 公文管理信息系统业务逻辑和数据处理层的实现	343
10.4.1 admin 表实体和对应会话 Bean	343
10.4.2 category 表的实体和会话 Bean	344
10.4.3 ofile 表的实体和会话 Bean	346
10.5 公文管理信息系统表现层的实现	348
10.5.1 登录页面	349
10.5.2 后台首页	351
10.5.3 添加公文	357
10.5.4 查看公文	360
10.5.5 修改公文	362
10.5.6 删除公文	365
习题 10	365
附录 A Java EE Web 编程技术教学大纲	366
附录 B 实验指导书	368
附录 C 使用日志记录	371
参考文献	374

第 1 章 Java EE 基础

本章主要介绍了 Web 应用的定义、Web 应用的体系结构、Java EE 的体系结构、几个典型体系结构例子、Java EE 应用程序的构成等，以便对 Java EE 的概念有一个基本的了解。Java EE 是功能强的 N 层应用系统规范，本章还叙述了 Java EE 的由来与发展（从 J2EE 到 Java EE）、Java EE 的新功能、集成开发环境 IDE（JBuilder、Eclipse、NetBeans 的比较）、编译和部署一个 JSP 页等，为后面章节的学习提供了 Java EE 概念的基础。

1.1 Web 应用基本概念

仅仅用了几年时间，在世界范围内，无论是信息的提供方式还是使用方式都因 Internet 而发生了改变，其硬件和软件技术使得每个人能够成为信息的使用者，而且几乎所有人都能够作为信息的提供者。Internet——特别是 World Wide Web（WWW），在非常短的时间内就已经被公众认为是重要的信息共享平台，许多组织都在尽力创建有用的 Web 应用，从而为用户提供更大的价值。

这些 Web 应用允许使用者在线购买图书和光盘。它们使得企业可以使用 Internet 来实施安全的事务处理。工人利用 Web 应用寻找工作；老板利用 Web 应用寻找雇员；使用由经纪人提供的在线应用可以购进和抛售股票；旅行者则可以利用 Web 应用来预订机票和酒店。这样的例子还有很多。很明显，目前无论是在公共的 Internet 上，还是在数不胜数的公司内部网（Intranet）中，都存在着大量有用的 Web 应用。

本书重点介绍基于 Java 平台企业版（Java EE）规范的企业 Web 应用所需的技术。

1.1.1 Web 应用定义

本书中，Web 应用有一个非常通用的定义——这是一种通过 Internet 技术加以连接的客户/服务器软件，可以传输其处理的数据。通过“Internet 技术”指的是在使用者和提供者之间，组成相应网络基础架构的硬件和软件的集合。Web 应用可以通过专门的客户端软件来访问，也可能利用一个或多个有关的 Web 页面访问，这些页面基于某种特定的用途可进行逻辑分组。这里所说的用途可以是任意一件事情，例如，可以是买书、处理股票订单，也可能仅仅是作为让使用者阅读的内容。

本书所讨论的是 Web 应用，而不只是“Web 网站”。实际上，这二者之间的区别对于理解本书的关键主题相当重要。大多数非技术人员往往不会区别 Web 网站和 Web 应用。除了术语的提法不同之外，对他们而言，其作用都是一样的，都可以使之实现在线购书、在线预订机票、在线购票等操作。不过，对于一名技术人员，两者之间还是存在差别的。如果有人谈到类似于一个 Web 网站的性能问题时，技术人员可能会开始想到其后台的具体细节，会考虑所运行的软件是 Apache 还是 IIS，还有它使用的脚本是 Java Servlet、PHP 或 CGI、perl。技术人员和非技术人员在思路上的不同可能会造成一定的误解。技术人员往往习惯性地“Web 网站”与服务器端联系起来。而与此同时，他们都知道，Web 应用并不

仅仅包括服务器端，它还需要有网络和客户端。因此，一个 Web 网站（服务器）与 Web 应用（客户机、网络和服务端）并不是一回事。

虽然本书所强调的是服务器端解决方案，在此还会讨论客户端和联网的有关内容，这是因为对于终端用户如何理解 Web 应用，它们有着很重要的影响。也就是说，会讨论 Web 网站内端对端的交互，它意味着由客户到服务器，再返回到客户，这是一个重点。毕竟，大多数使用 Web 的人所关心的就是它的端到端的操作。

可以说使用者是通过客户端软件（即使用 Web 来检索和处理数据的 Web 浏览器或应用）来使用 Web 应用的，这些客户端软件运行在客户端硬件（即 PC、PDA 等）之上，而应用数据的提供以及生产者对具体处理的控制则均通过服务器端软件（即 Web 服务器、服务器端组件软件、数据库等）来实现，这些服务器端软件运行于服务器端硬件（即高端多处理器系统、集群等），将客户端与服务器进行连接（从调制解调器或客户端设备的网络端口到服务器端的网络设备）就形成了联网基础架构。

在此，需要强调一点，即作为服务器软件的一种，要把 Web 服务器特别区别出来，因为它在调度客户端与服务器之间的通信（HTTP）时总是起着一个核心的作用。在本书中，当提到“服务器端”时，一般都包括 Web 服务器。

1.1.2 Web应用体系结构

1. 抽象Web应用体系结构

先不考虑具体情况，应用的体系结构必须能够体现业务逻辑、数据、接口和前面描述过的网络需求。实际上，在描述一个原型的应用体系结构时，最好是从一个非常通用的设计开始。然后再循序渐进地加入一些内容，如 Web 服务应该设置在哪里等。重要的是不要因特定的情况而迷失了方向。时间会改变、技术会改变，但是客户的需求基本上都保持不变。

2. 从客户到服务器

图 1-1 显示了一个非常抽象的应用组成，可以看到用户直接与接口交互，这样，接口需求也要在此满足。一个接口就是应用核心业务逻辑的代理，而业务逻辑则由一组操作组成，这些操作分别对应为手工完成的业务处理。执行此逻辑时，一般需要与数据交互，并对数据进行管理，就是要存储和查询数据。

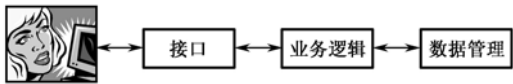


图 1-1 抽象应用体系结构

现在可以增加一些具体细节。一般来说，Web 应用包括一个通过网络与应用交互的用户。业务逻辑的执行可以在本地或远程完成。换一种说法，这表示客户可以很“胖”（本地逻辑），也可以很“瘦”（远程逻辑），一个胖客户和一个瘦客户的差别关键在于业务逻辑是在客户端或是在服务器端。无论是哪一种情况，至少接口不能远离用户，而且数据需要在某些服务器上集中。注意：接口总在客户端，而数据管理则总是在服务器端。

最后，还可以采用一种混合的设计，即业务逻辑既在客户端又在服务器端。实际上，

这种设计更为通用。数据验证即为一例，例如，要确保电话号码的形式如“123-456-7890”，尽管一般把它看做一个业务逻辑的问题，但是这种验证往往都通过类似于 JavaScript 等技术 in 客户端执行。

3. N层应用体系结构

应用体系结构一般都包括三种基本的组件：客户、网络和服务端。

1) 客户

常见的客户有两种类型：一种是人类客户，另一种是自动的、基于软件的客户。人类客户通常有一台带有操作系统的主机并能访问网络。一般使用一个 Web 浏览器来访问 Web 应用，不过也可能使用定制的接口。浏览器采用 HTTP 协议通信。人类客户的一个独特之处在于它与服务器的会话不要求持续的服务，由于人类客户往往会有许多“思考时间”，这使得服务器可以将资源分配给其他请求服务的客户。

较之人类客户所用的主机，自动客户可能运行在一个功能更为强大的主机上。自动客户可能使用 HTTP 协议，也可能使用低级或专用协议实现通信。这一类客户还有可能使用类似于消息等技术进行通信，这样就不会涉及 Web 服务器，但是却会涉及其余的服务器端软件（如应用服务器和数据库）。自动客户不需要“思考时间”，所以可能会不断地向服务器提出请求。

2) 网络

在客户和服务端之间的网络通常被称为 Internet，它是由分布在世界各地的许多主机和子网组成的。

主机相互通信时，其数据包将流经多个硬件和软件系统，并路由到最终目标。根据需要，其消息大多数使用 TCP/IP 协议完成通信。IP 代表网际协议，而 TCP 代表传输控制协议。TCP/IP 是一个面向连接的协议，它可以提供服务质量保证。即使底层网络并不可靠，但它也可以确保数据字节可靠地发送到通信双方。

因为 Internet 代表着一个不可靠的网络，而此协议几乎成为了它的默认“语言”。不过，仍要指出，还存在着其他的传输协议，最典型的是不可靠数据报协议（Unreliable Datagram Protocol, UDP）。此协议不是面向连接的，而且不能同样保证 QoS。不过，正是由于它不具备这些特性，反而使它比 TCP 效率更高。尽管 UDP 对于 Internet 应用来说一般是不能接受的，但是对于某些要求高性能的 Intranet 应用而言具有重大意义。

（1）客户端网络元素。客户通常并不直接与 Internet 相连。大多数人都通过一个服务提供商（称为 Internet 服务提供商）或 ISP（Internet Service Provider）来访问 Internet。在对 Internet 上的原始地址发出请求之前，客户往往会首先访问其浏览器缓存，查看是否已经有所需文档。浏览器缓存仅仅是客户主机的文件系统。请求一个页面时，如果它并不在本地的文件系统中，浏览器就会负责获取。提供此页面后，如果它是可以缓存的，则会在本地文件系统中保留一个副本以备日后访问。这样可以提高客户性能，因为如果没有此缓存，新的访问可能还需要消耗网络的往返时间。不过页面不会永远被缓存，因为它有到期时限，另外相应协议可以检查某个特定 Web 对象的更新情况。利用浏览缓存或其他缓存将远程 Web 页面数据进行本地存储（以避免为访问同一内容而要与原始服务器进行通信的开销），这一概念一般称为 Web 缓存。

代理缓存可以是软件也可以是硬件，它对频繁请求的 Web 页面进行缓存，这样 ISP 就不必重复地为其客户获取同样的页面。客户 A 首次获取 Web 页面 X 时，代理缓存会从原始

服务器请求页面，并将该页面的一个副本存储在其缓存中（假设此页面是可以缓存的），与此同时还会为客户 A 提供一个副本。当客户 B 请求页面 X 时，代理缓存只需从其缓存中获得页面，而不必再次访问网络了。这样就可以得到更好的客户性能。不仅如此，多个客户还可以分享此结果。图 1-2 显示了客户 Web 浏览器和中间的客户端缓存之间的关系。

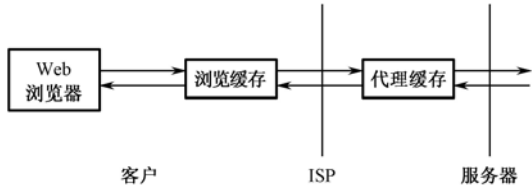


图 1-2 客户端网络基础架构

（2）服务器端网络元素。与客户端一样，服务器端也有一个 ISP，不过，可以对这里 ISP 的选择有所控制。了解带宽限制以及提供商可以访问的主干网络，在设计应用时这一点很重要。

关于网络服务器端部分还有另一个重要的方面，即把可能到来的连接分发给服务器资源的方式。本地的负载均衡器为实现多台主机之间工作负载的分摊，既提供了简单的方法，也提供了很复杂的技术。请求可以基于 Web 服务器的可用性被路由，或者也可以根据请求本身的实质进行分发。例如，图像请求以某种方式发送，而对静态页面的请求则采用其他方式发送。Cisco Local Director 就是一种典型的负载均衡硬件设备。

所平衡的负载通常在 Web 服务器场（Server farm）中进行分发。每个 farm 都由一组设计用来访问相同类型内容的 Web 服务器所组成。各 Web 服务器通常在单独的主机上，建立冗余可以提高 Web 站点的可靠性。

最后，可以在服务器端建立反向代理缓存，对于频繁访问的对象，通过提供对其快速访问来减少对服务器的请求。反向代理缓存的工作和客户端代理缓存的工作很类似，它会保存频繁请求对象。在服务器端这是很有用的，因为即使在多个客户的 ISP 并不相同的情况下，它也允许将经常访问的内容加以缓存。图 1-3 显示了这种部署策略，在此涉及一个负载均衡器、反向代理缓存和 Web 服务器场。

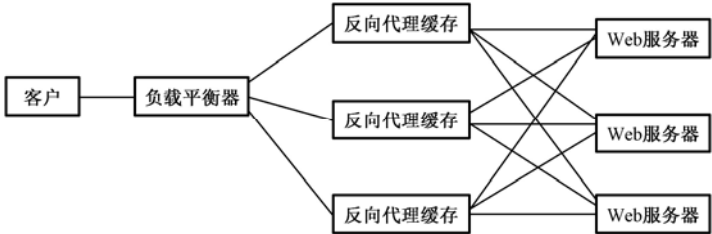


图 1-3 服务器端网络基础架构

（3）界于客户端与服务器端之间的网络元素。它们的存在是通过减少一些请求量特别大的资源的拥堵情况，提供一种通用而经济的服务。不过，如果了解中间缓存是如何工作的，以及 HTTP 协议的工作原理，那么至少可以向网络提供有关的数据的本质特性。有了足够的信息，中间网络元素就可以显著地减少网站上对于静态 Web 页面的负载。

一般来讲，客户端和服务器端之间的通信包括客户 ISP，它能够提供 T1 和 T3 速度，并通过主干网络服务提供商（Network Service Provider, NSP）转发请求。而 NSP 则以 OC-1

和 OC-3 速度实现通信。这些 NSP 提供了对网络接入点（Network Access Point, NAP）的访问，NAP 则是一些公共的交换设施，在此各个 ISP 可以通过一种称为 ISP 对等（ISP peering）的过程相互访问。NAP 分布在世界各个地方，将其合在一起，就代表把 Internet 主干“缝”在一起的所有点，即“针眼”。NAP 上的通信速度相当快，例如，可能达到 OC-12（622 Mbps 或更高），而且采用的是点对点的方式传输。

为了对客户与服务器的连接有更确切的体会，请参见代码清单：例 1-1，这是 traceroute 的输出，traceroute 是一个诊断型网络工具，可以跟踪由客户到服务器的数据包。下面代码中的数据包来自 Carnegie Mellon 大学，其目标为 Yahoo。

【例 1-1】代码清单 traceroute 输出，描述了 CMU 到 Yahoo 的路由情况。

```
1  CAMPUS-VLAN4.CMU.NET(128.2.4.1)1.744ms 1.052ms 0.992ms
2  RTRBONE-FA4-0-0.GM.CMU.NET(128.2.0.2)37.317ms 54.990ms 75.095ms
3  Nss5.psc.net(198.32.224.254)2.747ms 1.874ms 1.557ms
4  12.124.235.73(12.124.235.73)11.408ms 22.782ms21.471ms
5  gbr1-pl100.wswdc.ip.att.net(12.123.9.42)17.880ms 21.404ms23.662ms
6  gbr4-p00.wswdc.ip.att.net(12.122.1.222)13.569ms 10.793ms 11.525ms
7  ggr1-p370.wsw3dc.ip.att.net(12.123.9.53)11.814ms 10.948ms 10.540ms
8  ibr01-p5-0.stng01.exodus.net(216.32.173.185)12.872ms 20.572ms 20.885ms
9  dcr02-g9-0.stng01.exodus.net(216.33.96.145)29.428ms 10.619ms 10.550ms
10 csr21-ve240.stng01.exodus.net(216.33.98.2)10.998ms 32.657ms 19.938ms
11 216.35.210.122.(216.35.210.122)11.231ms 20.915ms 32.128ms
12 www7.dcx.yahoo.com(64.58.76.176)36.600ms 10.768ms 12.029ms
```

可以看到 CMU 是通过主干提供商 AT&T 和 Exodus Communications 与 Yahoo 建立连接的。

最近几年，对于客户和服务器之间这个领域的优化方法也得到了关注，即内容分发。内容分发（Content distribution）是从 Web 缓存中产生的，这是提供商用于复制其内容的一种方法，在复制时需要通过作为反向代理缓存的提供商，如 Akamai 等内容分发者对其内容进行策略性的复制，从而保证客户访问相当迅速，而且也不会涉及原来的服务器。内容分发的解决方案通常针对的是类似于图像等对带宽有压力的对象，这种对象即使不需要服务器端应用逻辑，也会很快消耗掉服务器端的带宽。

4. 服务器

服务器端应用体系结构不仅是最复杂的，而且也是最有趣的。通过对应用的客户和网络部分特性的了解并适当应用，可以从某种程度上对应用性能进行优化，而这些工作还会给服务器端带来更大的影响。可扩展性和性能方面存在的最大难题就在于此。图 1-4 显示了此体系结构的主要部分。从左到右，请求处理器通常是应用请求第一个到达的组件。

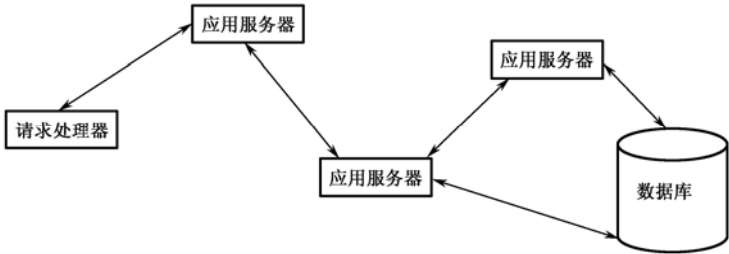


图 1-4 服务器端组织

请求处理器的一个例子即为 Web 服务器。Web 服务器有两个作用：对其能够处理的请求予以解决，如果不能解决则重新路由。对于静态的 Web 页面，Web 服务器可以通过访问文件系统来得到所需要的页面，从而在本地解决请求。作为一个请求路由器，Web 服务器可以确定请求的类型，并将它派给合适的处理程序，在本书中处理程序一般指的是一个 Java Servlet 引擎或 Servlet 容器。Servlet 容器会调用合适的 Servlet 实例，即某个 Java 类的一个方法。此 Servlet 将负责解开打包的请求参数，而且很有可能还将构造 HTML 响应。在此期间，它可能会完成许多工作，例如，与一个应用服务器或数据库建立联系以实现查询处理。

请求处理器的作用是识别请求的本质特性，并将其发送到一个实例上，该实例具有相应的功能可以执行所需的业务逻辑。这种机制通常被称为一个应用服务器。例如，应用服务器往往用于查找某个应用逻辑的实例。此逻辑的形式可以是一个 Java 类，并实现为一个企业 JavaBeans（Enterprise JavaBeans，EJB）或一个 CORBA 对象。EJB 和 CORBA 等技术均为中间件技术，可以通过应用服务器使应用逻辑更具健壮性、可扩展性和互操作性（应用服务器要对其进行管理）。

在一个设计合理的系统中，这些工作都是在后台完成的。应用逻辑的代码编写与应用服务器无关。它们明显的不同之处在于：一个提供访问，另一个提供业务功能。问题在于要为应用服务器提供自动管理业务逻辑部署的能力，在此要保证底层代码透明性的同时，还要为服务器机制提供最大的灵活性。本书后面将讲解 EJB 容器和 EJB 如何支持技术人员构建真正的应用服务器。

基于现有的应用数据，为了解决一个应用请求，数据必须在其所存储的位置被访问，这个位置一般是一个数据库。数据库要存储有关应用状态的信息，这包括用户、简表等所有作为应用数据组成部分的内容。但是，除了表示应用状态的数据以外，数据库还可以存储大量的业务逻辑。通过存储过程、触发器和数据库约束等形式即可达到此目的。数据库中的数据根据某种数据模型加以组织，这是一个表示数据应该如何存储的逻辑规范。在物理上，数据与数据表相关联。表数据（以及其他结构）则要写到磁盘上，这与文件系统中的数据完全类似。

注意对于应用服务器和数据库还可以有其他的客户，可能是基于消息的系统，也可能是原有的系统，它们使用如电子数据交换（EDI）等技术实现通信。这些客户往往是另外一些企业，而不是单独的用户，而且他们一般以批处理的模式与服务器端软件进行交互。

1.1.3 基于层的设计

在客户请示得到处理之前所经过的区域即称为层。每个层都与一个或多个逻辑相关联：表示、业务或数据访问，等等。2 层应用一般由表示层和数据层组成，例如，一个直接访问服务器端数据库的客户应用程序。3 层应用则更进一步：存在一个更瘦的客户端与某个 Servlet 或 CGI 程序联系，而后者再与数据库通信。表示层和数据层更明显地被分离，而服务器端也从数据访问逻辑中区别出了业务逻辑。一个 n 层应用则包括 3 级或更多的请求处理，其中包括数据库。对此可以举一个例子，一个客户与某个 Servlet 联系，而 Servlet 则与一组应用服务器通信，每个应用服务器均可以访问数据库。当前的许多 Web 应用系统都是 n 层的。

可能会感到疑惑， n 层的设计是不是有必要呢？其回报如何？直观地看，它使客户/服务器的连通性变得更为复杂。通信路径现在包括了客户→服务器 1→服务器 2→……→服务

器 $n \rightarrow$ 数据库。看上去好像一次发送需要更多的中继，这意味着更大的延迟，同时出现故障或瘫痪的机会也越多。是这种设计不太适合吗？答案是：只要设置得当，它就会带来很好的效益。

1) 提高模块化和组件的重用性

多个层意味着可以工作划分，并且可以将部分问题交给独立的模块完成。但是如果不做又会怎样呢？如果所有的工作都在单独的服务器端程序中完成，情况又如何呢？

例如，假设需要建立一个服务器端模块，支持 Web 用户预订图书。在编写此模块时需要确保当 Web 用户对其下订单，当按下“确认 (Confirm)”按钮时，应该完成以下事件：

- (1) 交互式客户请求的接收。
- (2) 信用卡检查。
- (3) 工作订单的创建。
- (4) 客户简表的更新。
- (5) 图书清单的更新。
- (6) 交互式客户响应的生成。

可以把所有这些操作都放在一个 **Confirm-and-Pay-for-a-Book** 模块中完成。但是假设两个月以后应用需求有所调整，需要处理 DVD 订单。想象一下会出现什么情况？需要创建一个新的模块，可以把它称为 **Confirm-and-Pay-for-a-DVD** 模块。以下是其所必需的操作：

- (1) 交互式客户请求的接收。
- (2) 信用卡检查。
- (3) 工作订单的创建。
- (4) 客户简表的更新。
- (5) DVD 清单的更新。
- (6) 交互式客户响应的生成。

是不是看上去很熟悉？不能重用订书模块中的功能。此时，可以创建一个更为通用的模块，称之为 **Confirm-and-pay-for-a-Product**，并以此来解决这个问题。

现在假设要求支持某种机制从而可以处理大量新产品的预订。假如，有一个销售商每天晚上把订单发给您，您需要处理这些订单。在这种情况下，无法简单地复用前面的通用产品模块，因为请求的类型基本上是完全不同的。只得重新创建一个新模块，它必须包括以下操作：

- (1) 接收批请求。
- (2) 对于每个请求进行以下操作：① 信用卡检查。② 工作订单的创建。③ 客户简表的更新。④ 产品清单的更新。
- (3) 生成一个批总结响应。

对于未经精心设计的模块和组件，无法复用其中的独立操作。这一点完全与程序设计类似，如果不能很好地进行模块化，应用的复用性也很小。因此，往往不得不对功能进行复制而造成代码迅速膨胀起来，这也常常导致应用操作的不一致性。

模块和组件的可扩展性还可能存在死角，由于没有复用功能，需要将它复制到多个模块中，这样就会消耗掉服务器主机上的内存，而这是完全没有必要的。相反，如果将这些模块和组件分解为细粒度的组件，就不仅能够复用其功能，而且还可以有效地消除

性能瓶颈。

2) 更好的分布式处理和任务并行化

进一步的模块化也带来更多选择。假设利用某种技术（如 Java RMI 或 CORBA）来连接这些模块，很有可能将每个更小的模块部署在不同的主机上。例如，可以让所有信用卡检查均在一个主机上完成，而所有数据库更新则在另一台主机上实现。这是一个很有诱惑力的解决方案，因为它允许将处理负载分配到不同的主机上。

在某些情况下，因为某个操作与其他所有操作是无关的，这样模块化就可以提供更好的性能。考虑一下订书例子中更新客户简表的操作。更新一个简表从概念上不会返回任何有用的信息。这只是一个必要的过程。在简表更新之后是对产品清单的更新，但并不需要等待简表更新完成才能进行处理。从理论上说，产品清单的更新可以与简表的更新并行处理。

通过将独立模块的处理交由不同的主机完成，可以提高执行期间的并行性，而更好的并行性也意味着提供更好的性能。当然，要有效地对这些任务进行并行化，需要建立一种异步的通信解决方案，也就是说，无需一个中间响应。只要做到了这一点，就可以通过提高任务并行化的程度使应用性能有显著的提高。

3) 更有效的服务复制

除了可以将组件部署在多个主机上外，还能够采用更小粒度的控制来解决可扩展性和性能问题。也就是说，可以确定解决方案，并很可能找出一个特定的系统问题。在所有应用系统中，一个常见的问题就是大型任务中的一个或多个子任务速度很慢，并成为整个任务的瓶颈。

下面再回到订书的例子中。假设信用卡检查每个请求需要花费 3 s，而需要一次处理 100 个订书单。现在只有服务器主机上运行有一个应用服务，而且该服务是单线程的。这说明：

- ① 最快的信用卡检查至少需要 3 s。
- ② 最慢的信用卡检查至少需要 $(100 \times 3) = 300 \text{ s}$ (5 min)。
- ③ 平均的信用卡检查至少需要 $(50 \times 3) = 150 \text{ s}$ (2.5 min)。

这种速度是无法接受的。也许可以将服务复制 100 次，同时为 100 个客户提供服务。那无论是最快、最慢，还是平均的时间都将为至少 3 s。

作为一个例子，假设大型的 Confirm-and-Pay-for-a-Product 模块的一个副本需要 5 Mb 的存储空间（代码本身需要量要小一些，但是要作为一个应用服务则需要这么多）。将该模块复制 100 次，即使有足够的硬件来支持，也意味着对于 100 个同时使用此功能的客户，要正常处理就需要 500 Mb 的内存。

与之相比，假设信用卡操作本身只需要 5 Mb 中的 1 Mb。通过把 Confirm-and-Pay-for-a-Product 模块分解为更小的独立分布式组件，并把信用卡检查等操作放在不同的组件中，就可以限制只复制必要的部分。例如，可以把信用卡检查操作复制 100 次，这样就只需要 100 Mb 的空间，而不是 500 Mb。

显然，细粒度的业务逻辑处理所具有的空间效率的优势使得 n 层部署更具有吸引力。如果在应用中存在瓶颈，就可以找出相关的组件，并在必要情况下进行复制（就像在高速公路上增加车道数量一样）。组件划分得越细，此复制的开销（按内存量计算）也就越少，相

应地，应用也能更好地得到扩展。

1.2 Java EE规范

Sun Microsystem 的 Java 平台企业版（Java EE）规范是为建立 n 层应用而提出的一种基于 Java 的解决方案。Java EE 的重点在于定义客户端和服务端技术，从而不仅使得应用更易于建立，而且也更容易得到集成。Java EE 规范中覆盖了各种客户/服务器类型及其交互：可以用于处理 Web 的信息服务器、纯应用服务器、Applet，另外无论是同步还是异步的解决方案也都能适用。

尽管 Java EE 提出了一些相当复杂且功能强大的技术，但它仍是一个规范，需要由供应商来具体实现开发规范所提出的功能。Sun 也发布了一个参考的实现，以此来了解 Java EE 不仅很有用，而且也很合适。在一些重要的细节上，开发商们往往存在着分歧。

1.2.1 什么是Java EE

什么是 Java EE？先应了解 Java2 计算平台。Java2 计算平台以 Java 语言为中心，其体系结构与平台无关，共有 3 个独立的版本，每一种版本都针对特定的产品类型。

(1) Java2 Platform, Micro Edition (J2ME): J2ME 针对消费产品市场。例如，移动电话、PDA、能够接入电缆服务的机顶盒，以及其他具有有限的连接、内存和用户界面能力的设备。J2ME 使得制造商和内容创作者能够编写适合消费市场的可移植 Java 应用程序。

(2) Java2 Platform, Standard Edition (J2SE): J2SE 针对包含丰富的 GUI、复杂逻辑和高性能的桌面应用程序。J2SE 支持独立的 Java 应用程序，或者与服务器进行交互的客户端应用程序。

(3) Java 平台企业版（Java EE）: Java EE 针对提供关键任务服务的企业应用程序，这些应用程序必须是高度可伸缩和可用的。Java EE 是基于模块和使用 Java 语言编写的可重用软件组件，运行在 J2SE 之上。

1.2.2 Java EE的体系结构

Java EE 是一个标准的多层体系结构，适用于开发和部署分布式的、基于组件的、高度可用的、安全的、可伸缩的、可靠的和易于管理的企业应用程序。Java EE 体系结构的目标是减少开发分布式应用程序的复杂性和代价，以及简化开发和部署过程。

Java EE 平台的简要体系结构如图 1-5 所示。Java EE 平台包含了创建一个标准 Java 企业应用程序的体系结构的程序设计模型，而这样的 Java 企业应用程序可以从客户层的用户界面跨越到企业信息系统（Enterprise Information System, EIS）层的数据存储。

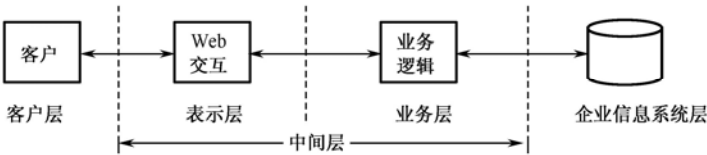


图 1-5 Java EE 体系结构

Java EE 体系结构是一个多层的、端到端的解决方案，这个系统结构横跨客户层到表示

层、业务层，最终到达企业信息系统层。Java EE 体系结构将一个企业应用程序分为客户层、表示层、业务层和数据层，这些层被映射到 Java EE 体系结构实现中处理特定功能的 4 个不同的层次。

(1) 客户层：通常是一台桌面计算机，客户可以使用 GUI 与应用程序进行交互。

(2) 中间层：由表示层和业务层组成，通常由一个或者多个应用服务器组成。应用服务器处理客户的请求，执行复杂的表示形式和业务逻辑，然后将结果返回给客户层。Java EE 应用服务器提供两种类型的应用程序框架和网络基础结构，它们被称为容器。容器为 Java EE 平台支持和两种类型组件提供运行时环境——Web 容器和 Enterprise JavaBean (EJB) 容器。

(3) 企业信息系统层：也称为数据层，是驻留业务数据的地方。在处理业务逻辑时，由中间层访问 EIS 层（如 ERP 等）。

1.2.3 Java EE应用程序构成及应用

1. Java EE应用程序构成

Java EE 技术提供了一个基于组件的方法来设计、开发、装配和部署企业级应用程序。Java EE 平台提供了一个多层结构的分布式的应用程序模型，该模型具有重用组件的能力、基于扩展标记语言 XML 的数据交换、统一的安全模式和灵活的事务控制。开发人员不仅可以比以前更快地发表对市场的新的解决方案，而且独立于平台的基于组件的 Java EE 解决方案不再受任何提供商的产品和应用程序编程界面（API）的限制。

1) Java EE 组件

Java EE 应用程序由组件组成。一个 Java EE 组件就是一个自带功能的软件单元，它随同相关的类和文件被装配到 Java EE 应用程序中，并实现与其他组件的通信。Java EE 规范是这样定义 Java EE 组件的：

- ① 客户端应用程序和 Applet 是运行在客户端的组件。
- ② Java Servlet、JSP 和 JSF 是运行在服务器端的 Web 组件。
- ③ Enterprise JavaBeans (EJB) 组件是运行在服务器端的商业软件。

Java EE 组件用 Java 编程语言写成，并和用该语言写成的其他程序一样进行编译。Java EE 组件和标准 Java 类的不同点在于：它被装配在一个 Java EE 应用程序中，具有固定的格式并遵守 Java EE 规范，它被部署在产品中，由 Java EE 服务器对其进行管理。

2) Web 组件

Java EE 的 Web 组件既可以是 Servlet，也可以是 JSP、JSF 界面。Servlet 是一个 Java 编程语言类，它可以动态地处理请求并做出响应。JSP 页面是一个基于文本的页面（JSF 是一个基于组件的用户界面），它以 Servlet 的方式执行，但是它可以更方便地建立静态内容。在装配应用程序时，静态的 HTML 页面和 Applet 被绑定到 Web 组件中，但是它们并不被 Java EE 规范视为 Web 组件。服务器端的功能类也可以被绑定到 Web 组件中，与 HTML 页面一样，不被 Java EE 规范视为 Web 组件。

一个 Java EE 应用程序可能包含一个或多个 Enterprise JavaBeans、Web 组件，或应用程序客户端组件。其中，应用程序客户端组件是运行于可允许其存取 Java EE 服务的容器（环境）中的 Java 应用程序。

3) Java EE 容器

容器是一个组件和支持组件的底层平台特定功能之间的接口。在一个 Web 组件、Enterprise Bean 或者是一个应用程序客户端组件可以被执行前，它们必须被装配到一个 Java EE 应用程序中，并且部署到它们的容器里。

装配的过程包括为 Java EE 应用程序中的每一个组件以及 Java EE 应用程序本身指定容器的设置。容器设置定制了由 Java EE 服务器提供的底层支持，这将包括诸如安全性、事务管理、Java 命名目录接口（JNDI）搜寻以及远程连接。

4) 容器类型

部署时会将 Java EE 应用程序组件安装到 Java EE 容器中，如图 1-6 所示。

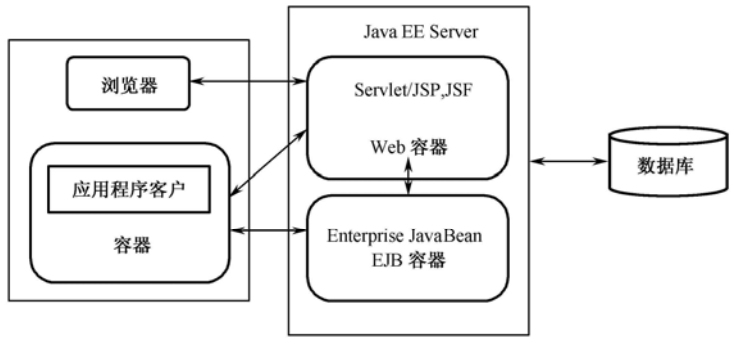


图 1-6 Java EE 服务器和容器

Java EE 服务器是 Java EE 产品的运行部分，它提供 EJB 容器和 Web 容器。EJB 容器管理 Java EE 应用程序的 Enterprise JavaBean 的执行。Enterprise JavaBean 和它的容器运行在 Java EE 服务器中。

Web 容器管理 Java EE 应用程序的 JSP、JSF 页面和 Servlet 组件的执行。Web 组件和它的容器也运行在 Java EE 服务器中。

客户端应用程序容器管理应用程序客户端组件的运行。应用程序客户端和它的容器运行在客户端中。

Applet 容器管理 Applet 的执行。它由运行在客户端的一个 Web 浏览器和 Java 插件一同组成。

图 1-7 是 Java EE 设计模式体系更详细的示意图。图 1-7 左边为客户层，有瘦客户（浏览器）、胖客户，即应用程序，包含 Java 应用程序、非 Java 应用程序、CORBA（公共对象请求代理结构）和 COM（组件对象模型）。中间层包括 Web 容器和 EJB 容器，它们合在一起代表了 Java EE 应用服务器，Web 容器和 EJB 容器分别用于处理表示和业务逻辑；代表业务层的 EJB 容器包含会话 Bean、实体 Bean 和消息驱动 Bean；中间层可由一个或多个应用服务器组成，应用服务器处理客户的请求，执行复杂的表示和业务逻辑，然后将结果返回给客户层。图 1-7 右边的 EIS（Enterprise Information System，企业信息系统）层代表数据层，包含 RDBMS、ERP、大型机事务处理（mainframe transaction processing）及其他遗留信息系统（legacy information system），在处理业务逻辑时，由中间层访问 EIS。另外，Java EE 在通信、安全、表示、业务应用以及企业信息系统方面均提供了相应的技术支持。

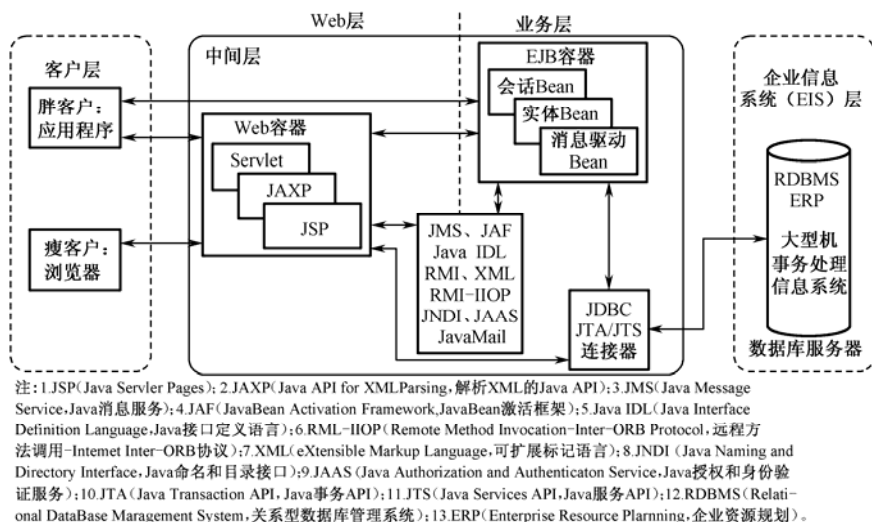


图 1-7 Java EE 设计模式体系示意图

2. 平台技术与服务例子—JNDI

在 Java EE 基础架构上建立应用有一大好处，就是可以充分继承一些关键的技术，从而保证可扩展的部署。了解一些 Java EE 基础架构提供的主要平台技术和服务是有益的。这些技术与分布式应用组件的简化和设置有关，目的是使组件能够进行协调和通信。在一个分布式系统中提供这种透明性，可以使应用的设计人员将重点放在其应用的业务逻辑上，而不会把时间浪费在复制的性能优化调整上，也不会浪费时间来原因该逻辑的组件分配。在这一节中，将了解 Java EE 基础架构提供的主要平台技术和服务的例子——实现资源查找的 JNDI。

尽管 Java EE 假设了一个分布式应用系统，一般仍有必要在系统组件之间维护一个公共的状态。例如，资源、命名或地址在不同组件之间都可以是公用的（即与位置无关）。对此状态的访问和管理可以由 Java 命名和目录接口（Java Name and Directory Interface, JNDI）提供。

对于熟悉目录服务的人来说，JNDI 是一种用于访问如 LDAP 等服务的方法。对于大多数 Java EE 开发人员而言，JNDI 则仅仅是一种用于管理多层共享资源的简单方法。

从组件的角度看，JNDI 为公共应用状态或上下环境（comtext），它提供了一种与位置无关的访问方法。Java EE 规范也将 JNDI 环境认为是一种特定的机制，从而使应用组件具有动态性，无须为某个组件调整其源代码，而是根据访问环境所获得的值来执行组件中相应的部分，而动态性正是基于此实现。在运行阶段，当组件访问其环境时，变量设置（即变量或规则）即将实现。这个工作可以在部署的基础上完成。也就是说，部署人员能够在部署描述中设置适合的值，组件通过 JNDI 来得到这个值。以上工作完成不需要重新编译。

例如，假设要定制一个倍数作为提高工资的基础，如果该值与部署有关，或者存在动态性，那么应用组件就需要从 JNDI 中读取，并完成适当的操作。

以下应用代码段所做的工作即实现一个虚设的函数，其函数名为 calculateNewSalary()。下面介绍的方法需要将以前的工资值作为参数，并使用 raiseMultiple 的值来确定新的工资。

【例 1-2】 JNDI 中读取 raiseMultiple 的值来确定新的工资。

```

Public double calculateNewSalary(double a_origSalary) {
    /*Obtain handle to our naming context.*/
    Context initCtx=new InitialContext();
    Context localCtx=(Context)initCtx.lookup("java:comp/env");

    /*Look up the salary multiple*/
    double raiseMultiple=((Double)
        localCtx.lookup("raise_multiple")).doubleValue();

    /*Raise salary per the multiple*/
    return raiseMultiple*a_origSalary;
}

```

这样，我们首先需要初始化环境，并在上下环境树结构中找到所需要的节点。一旦找到，则读取该目录树的一个叶子——`raise_multiple` 作为值。例如，10%的增长表示这个值为1.10，可以将该值作为计算新工资的基础。

在部署描述符中，`raise_multiple` 必须做如下存储：

```

...
<env-entry>
    <env-entry-name>raise_multiple</env-entry-name>
    <env-entry-type>java.lang.Double</env-entry-type>
    <env-entry-value>1.10</env-entry-value>
</env-entry>
...

```

JNDI 的其他用途是对各种共享 Java EE 资源提供访问，这其中包括：数据库、消息队列、EJB、事务、邮件服务器、URL。

使用 JNDI 来查找对象，这样对于获得针对某个公共资源的访问，就可以提供简单一致的接口。

下面举例说明如何通过 JNDI 从环境中获取这些对象，可以考虑如何找到一个已知的 Java 消息服务（Java Message Service, JMS）队列。尽管其名字是已知的（即队列就以此名在 JNDI 中注册），其位置却是未知的。这就体现出了 JNDI 价值：并没有让每一个应用组件都了解各个资源所在的物理位置，J2EE 对分布式资源提供了一种逻辑控制。这样就使得对应用的组件的编码更加简单，而且不需要重新编码或重新编译使用该资源的应用组件，就可以完成组件的重定位。

以下代码显示了如何利用 JNDI 来获得一个已知的消息队列的句柄。

【例 1-3】用 JNDI 来获得一个已知的消息队列的句柄。

```

/*Obtain handle to our naming context.*/
Context initCtx=new InitialContext();
Context localCtx=(Context)initCtx.lookup("java:comp/env");
/*Look up the message queue we want*/
Queue myQueue=(Queue)localCtx.lookup("jms/MyQueue");

```

由于 `raise_multiple` 的值是在部署描述符中定义的，因此与 JMS `MyQueue` 对应的信息同样也放在部署描述符中。注意对于这两种情况，资源类型和描述也可以放在描述符中，这就

使环境具有了自描述性。代码如下：

```
<resource-env-ref >
    <description>
        My message queue
    </description>
    < resource-env-ref-name >jms/MyQueue</ resource-env-ref-name >
    < resource-env-ref-type >javax.jms.Queue</ resource-env-ref-type >
</ resource-env-ref >
```

1.2.4 几个典型Java EE体系结构

1. 可选Java EE不同体系结构

现在或许会感到疑惑，需要使用所有的 Java EE 技术吗？如果没有必要，那么究竟应该使用多少 Java EE 技术呢？答案是，“只要想用应尽可能地使用”。

为了证实这一点，实际上，大多数 Java 开发中都已经使用了 Java EE 体系结构中的一些通用部分，如 Servlet、JSP 和 EJB 等。JDBC 已经成为一个很常用的机制，从而可将数据库集成到 Java 应用中。最近，随着人们再一次认识到异步通信的妙处，JMS 也得到了普遍关注。Java EE 的优点在于如果想使用部分技术，并不需要将整个 Java EE 都包括在应用中。例如，可能不需要使用 EJB，而只是使用 Servlet 和 JSP。或者，也可以只使用其他，均可以根据需要来加以选择。

不过当确实要使用 Java EE 时，还意味着需要选择一个开发商。要求开发商既保证符合 Java EE 规范，同时还能提供相应实现方法以满足需要。例如，由于为 EJB 实体 Bean 实现容器管理持续存储（自动数据管理）的方法不同，开发商之间也存在很大差异。这就是选择开发商之前所必须了解的细节。

最后，暂不考虑具体方法，如果对建立一个具有高可扩展性的 Web 应用感兴趣，那么最好考虑 n 层方式。在建立这些层时，会发现部分甚至全部 Java EE 规范将会很适合于满足要求和计划。

2. 几个典型体系结构例子

虽然不可能存在一种适合所有应用的软件体系结构，但是通常可以复用一些通用的体系结构模式。

本章前面曾讲过，Java EE 平台使开发者能够方便地根据不同的配置创建不同的 n 层（多层）应用程序体系结构。基本的 n 层体系结构在每层都可能有一定的数目的组件，并在层与层之间建立交互。

下面简要介绍一下在开发基于 Java EE 系统时可能用到的（或遇到的）一些体系结构。其中每种体系结构都有自己的优点或长处。下面举例说明了几种应用程序的构成，便于读者理解这些不同的体系结构。

1) 应用客户端和 EJB

如图 1-8 所示的体系结构，应用的客户端由表示层组成（应用客户端是一个基于 Swing 或者 AWT 的 Java 应用程序，或者是一个控制台程序），并且与业务层的 EJB 进行通信。



图 1-8 应用客户端和 EJB 体系结构

客户端应用是一个单机的（JFC/Swing 或控制台）应用，这个应用依赖于另外一台计算机上 EJB 所实现的业务规则。

2) JSP 客户端和 EJB

如图 1-9 所示是基于 JSP 的体系结构。在此体系结构中，应用客户端是在浏览器中显示的一个 Web 页面。服务器上的 JSP 直接与业务逻辑层进行交互，响应客户端浏览器发来的请求，然后产生 Web 页面并将该页面发送到客户端浏览器。

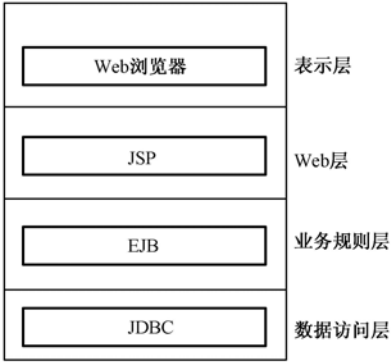


图 1-9 JSP 客户端和 EJB 体系结构

实际上，这个体系结构中的客户端就是 Web 浏览器。JSP 访问业务规则并且为客户端浏览器产生页面的内容。

3) Applet 客户端和 JSP 及数据库

图 1-10 所示的体系结构类似于图 1-9 中的体系结构。这里的客户端是一个表示层的 Java Applet，它通过网络可以和业务层的 JSP（或 Servlet）进行通信。像图 1-9 中的例子一样，业务逻辑层也可以是 EJB，但是本例中业务层由 JSP 构成。

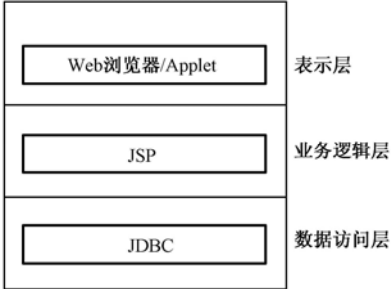


图 1-10 Applet 客户端和 JSP 及数据库体系结构

在 Web 页面中，Java Applet 展示给用户的图形界面更具交互性和动态性。Applet 从 JSP 获取额外的内容。即使通常是用 JSP 产生 HTML 的 Web 页面，JSP 也可以只由业务逻辑构成。此外，JSP 使用 JDBC API 从数据库访问数据。

4) 应用集成的 Web 服务

尽管 Java EE 是基于 Java 的，但是 Web 应用体系结构却不只限于使用 Java 组件，如数据层，有许多企业级的数据库是用 C 或 C++这样的高级语言来实现的。类似地，对于某一层的组件而言，只要能提供明确定义的接口且可以进行互操作通信，那么也可以使用 Java 之外的语言实现。如图 1-11 所示的例子，这是一个用 C#语言实现的客户端应用访问一个用 Java 实现的 Web 服务。

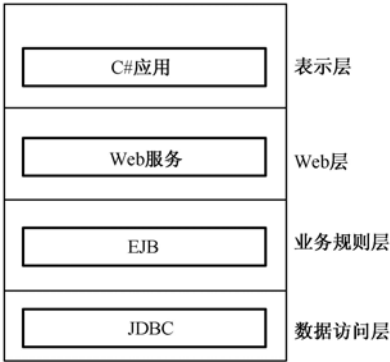


图 1-11 应用集成的 Web 服务体系结构

Web 服务接口在客户端和 Web 服务之间提供了定义良好的接口，可以用任何一种支持发起 HTTP 请求的语言来实现客户端。例如，用 C#语言编写客户端程序，然后由 Java 编写 EJB；客户端向应用服务器发送合乎格式的 Web 服务请求，Java EE 应用服务器运行 EJB 来为客户端提供服务。

1.3 Java EE Web应用的编译和部署

1.3.1 Java EE的部署问题

本节中不打算介绍如何运行 Java EE 的某种具体实现，也不会针对不同开发商关于 Java EE 所做调整的细节进行讨论。实际上，一般会忽略详细的部署问题，因为这很大程度上取决于开发商，而且相对于体系结构和设计，它与 Java EE 操作的关系更为紧密。不过，在此简要说明 Java EE 应用是如何包装和部署的还是很有帮助的。在后面的讨论中，如果指出某个问题属于部署问题，那么它往往就与这里所介绍的某个（或某几个）方面有关。

1. 包装

所谓包装（packaging）就是将 Java EE 应用存储在一个带有.ear 扩展名的文件中（EAR 文件）。EAR 文件与.jar（JAR）文件是相同的，只不过它是一个企业 Java 应用归档文件，而不仅仅是一个 Java 应用归档文件。另一种重要的文件类型是 Web 应用归档文件，或.war（WAR）文件，它将把与 Servlet 及其 JSP、JSF 页面有关的文件进行归档。

一个 EAR 文件可以包括一个或多个以下内容：

- (1) 对应每个 EJB 组件的 JAR 文件。
- (2) 对应每个应用客户的 JAR 文件。
- (3) 对应每个 Servlet 和一组相关 JSP、JSF 的 WAR 文件。

EJB 和应用客户 JAR 包括有关的.class 文件，还包括所有相关的文件，另外还包含有一个部署描述符。WAR 文件包括 Servlet.class 文件，还带有 GIF 文件等静态 Web 资源，另外也包括一组相关的文件（即工具 Java 类）以及一个部署描述符文件。

为了对 EJB 的 JAR 文件有比较具体的了解，下面来看一个名为 BenefitSession 的示例 EJB 的 JAR 文件。

【例 1-4】 一个名为 BenefitSession 的示例 EJB 的 JAR 文件内容。

```
BenefitSession/  
    EjbBenefitSessionbean.class  
    EjbBenefitSessionRemote.class  
    EjbBenefitSessionHome.class  
META-INF/  
    Ejb-jar.xml
```

所装配的内容以及所部署的内容之间存在着一定的关系，如图 1-12 所示。可以看到，WAR 和 JAR 文件均为 Java EE 应用 EAR 文件的组成部分，而 Java EE 应用 EAR 文件则部署为一个 Java EE 服务器。

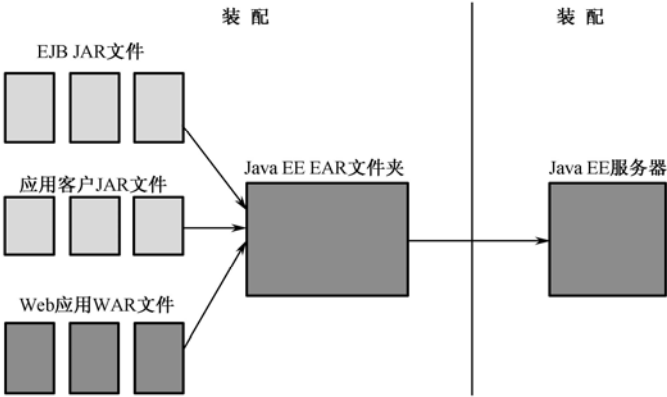


图 1-12 Java EE 服务包装和部署

2. 部署描述符文件

部署描述符文件是一个 XML 文件，其中通常包括有关 Java EE 组件的结构化、装配和运行时的信息。一个 Java EE 应用包括一个针对整个应用的部署描述符，另外还会在必要情况下为每个组件指定部署描述符文件。部署描述符文件中的信息必须处理组件的装配（如 EJB 接口类的名字）、安全信息（如角色的定义和应用）以及其他运行时相关性和信息。

在此显示了一个示例 EJB 部署描述符文件的部分内容，其中指出了有关对象的结构（即其属性和关系）和其资源相关性（即数据库信息）。

【例 1-5】 一个示例 EJB 部署描述符文件的部分内容。

```
<persistence-type>Container</persistence-type>  
<cmp-field><field-name>first_name</field-name></cmp-field>  
<cmp-field><field-name>last_name</field-name></cmp-field>
```

```

<cmp-field><Field-name>hire-date</field-name></cmp-field>
<resource-ref>
    <res-ref-name>employee</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>

```

无须深入了解有关部署描述符组成的详细内容，也可以看出它仅是一个 XML 文件，其中包括了关于 EJB 或所部署的应用组件的元数据。作为一个 XML 文件，部署描述有助于确保部署的可移植性，即使采用不同的部署平台，也可以使用同一组属性。毕竟 XML 文件只不过是一个文本文件，它与操作系统和 CPU 无关。

以例 1-6 为例说明部署描述符如何包括关于安全角色和特权的信息。

【例 1-6】 示例部署描述符。

```

<ejb-jar>
...
<enterprise-beans>
...
    <entity>
        <ejb-name>MyBean</ejb-name>
        <ejb-class>MyEntityBean.class</ejb-class>
        ...
        <security-role-ref>
            <role-name>entity-admin</role-name>
            <role-link>administrator</role-link>
        </security-role-ref>
    ...
    </entity>
    ...
</enterprise-beans>
...
<assembly-descriptor>
    <security-role>
        <description>The Admin Role</description>
        <role-name>administrator</role-name>
    </security-descriptor>
    ...
</ejb-jar>

```

注意：新的安全角色可以定义，也可以进行链接，它能够应用在如 EJB 等不同的应用组件中。

Java EE 部署描述符的完整 XML DTD 可以在 http://java.sun.com/dtd/application_1_3.dtd 中找到。图 1-13 以图的形式对 DTD (DTD 详见 8.2.2 节) 做了总结，这几乎是未加改动地直接选自 Java EE 规范，它将有助于了解一个 Java EE 应用是如何分解，以及如何组织装配和部署信息的。

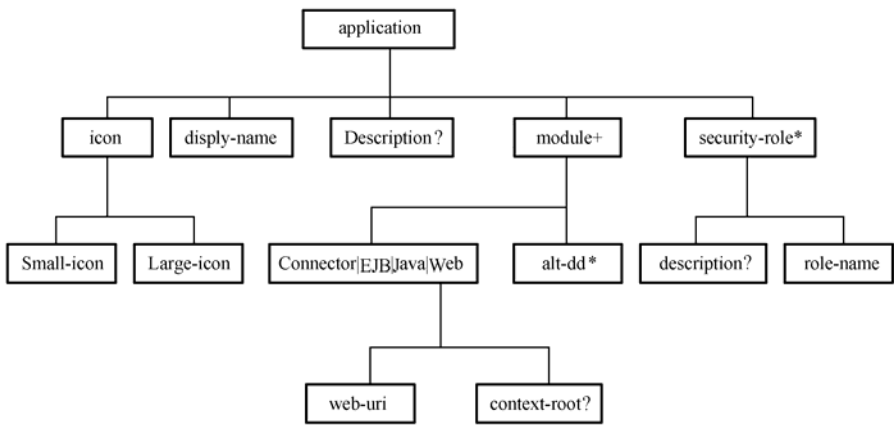


图 1-13 Java EE 部署描述符 XML DTD

如图 1-13 所示，一个部署描述符包括了关于各种类型应用信息的信息，也包括其模块。注意此 DTD 是关于整个应用的（即 EAR 文件）。应用中各元素，如 EJB JAR 文件和 WAR 文件则有其自己的部署描述符 DTD。

同样的，本书不考虑针对每个组件的部署描述符所做的详细调整。在讨论 JMS 和 EJB 等特定 Java EE 技术的书中，对此都提供了大量的信息，在这方面它们是最好的资源；Java EE 开发商文档也是一个很好的资源，其中将提供有关部署最为具体的信息。对于部署描述符设置做更多的评论可能很枯燥，而且也偏离了本书的中心。本书不做讨论，有兴趣的读者可自行查阅。

1.3.2 创建一个JSP应用程序的实例

1. 编译和部署一个JSP页面

下面通过一个 JSP 页面实例了解部署的大概过程。此过程包含若干步骤，经过这个过程不但可以确认 Java EE 服务器是否正常，而且还可以体会如何建立、部署和测试 Java EE 应用程序。

(1) 创建一个工作目录。在这个工作目录里创建和编辑应用程序文件。

(2) 创建一个 JSP 页面的文本文档。这是一个 HTML 格式的文本文档，其中插入一些 Java 代码段，Java EE 服务器会将这个编译生成 Servlet。

(3) 把创建好的文件都打包到一个 Web 文件（Web Archive，WAR）中，WAR 是一种 JAR 格式的文件，为了方便部署，它把所有应用组件都打包到一个单一文件中。

(4) 把 WAR 文件打包成企业文件（Enterprise Archive，EAR），其中包含一些应用服务器（如 JBoss）的部署指令。

(5) 把 EAR 复制到应用服务器（如 JBoss）的部署目录中去。一旦完成这步，就可以运行应用程序了。

(6) 测试这个应用程序。

2. 创建步骤

(1) 在计算机上创建一个目录，用于存放练习。本例中建立目录 C:\BJEE5\Ch02。

(2) 用文本编辑器在刚才的目录下创建一个新的文本，文件名为 index.jsp。

```

<%--
file: index.jsp
desc:Test installaataion of java EE SDK 5
--%>
<html>
<head>
<title>Hello world - test the Java EE SDK installation
</title>
</head>
<body>
<%
    for ( int i = 1; i < 5; i++)
    {
%>
        <h<%=i%>>Hello world</h<%=i%>>
<%
    }
%>
</body>
</html>

```

(3) 创建一个子目录 **META-INF**，并在这个目录里创建一个名为 **application.xml** 的文件。该文件包含了一些设置，这些设置用于标识应用程序和所依赖的 **JBoss** 资源，并且配置用户将如何访问这些资源。该文件的内容如下所示：

```

<?xml version= "1.0"?>
<application>
  <display-name>Hello Java EE world!</display-name>
  <module>
    <web>
      <web-uri>web-app.war</web-uri>
      <context-root>/hello</context-root>
    </web>
  </module>
</application>

```

(4) 需要创建一个新的 **WAR** 文件和 **EAR** 文件。**WAR** 文件包含 **Java EE** 应用的 **Web** 组件，并用一个描述文件或“内容列表”来说明这个文档的内容。通常，一个 **Web** 应用包含的文件数量要比这个简单的例子多很多。为了易于发布，可把所有的文件都打包到一个 **WAR** 文件中。

```

>jar cf web-app.war index.jsp
>jar cf helloworld.ear web-app.war META-INF

```

(5) 把 **EAR** 文件 **helloworld.ear** 复制到 **JBoss** 应用服务器的部署目录中 (**C:\jboss\server\all\deploy**)，然后从命令行启动 **JBoss** 应用服务器。

```

C:\>%JBoss_HOME%\bin\run -co all

```

(6) 测试第一个 **JSP** 页面。启动一个 **Web** 浏览器输入下面的 **URL** 地址：**http://localhost:**

在数秒之后，即可看到一个 Web 页面。

本例创建的 JSP 文本文件是由 HTML 和内嵌的 Java 代码段构成的，应注意文件中的 Java 代码外围的标记（详细内容见第 4 章）。

1.4 Java EE的发展与特点

1.4.1 Java EE的由来与发展

美国 SUN 公司的 Java 平台企业版（Java Platform, Enterprise Edition）规范是 J2EE 1.5 版本之后的称谓（简称 Java EE）。纵观 Java EE 规范的历史可以看出，每次重大修订都是由一个重要主题推动的。例如，第一次发布 J2EE 1.2 时，伴随的重要主题是首次将单独的规范绑定在一起，后来，在 J2EE 1.4 中，关注的重要主题则是 Web 服务。图 1-14 显示了 Java EE 的摘要历史，列出了每个版本的重要功能以及促成每次修订的一些重要外部影响。

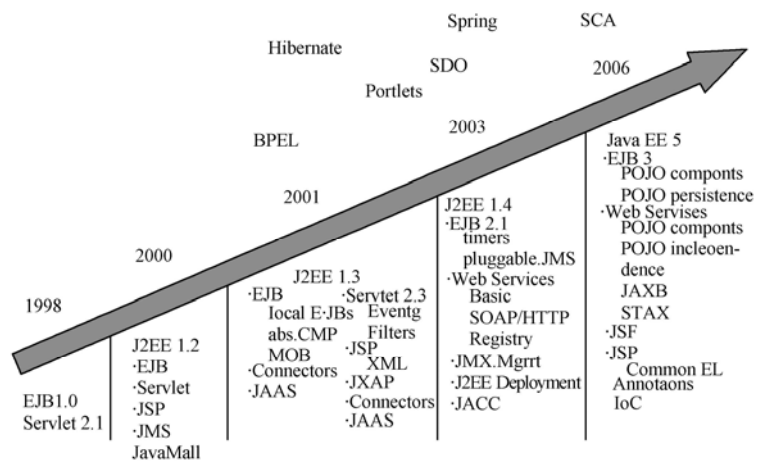


图 1-14 Java EE 的历史

与一些新技术的大多数早期版本一样，Java EE 规范的以前版本中存在一些“难点”，其中包括：

- (1) 业务逻辑编程的复杂性。
- (2) 持久性编程模型的复杂性和性能。
- (3) 表示层/逻辑混合。
- (4) Web 服务的类型、复杂性、文档模型、扩展和性能。
- (5) 多成员团队开发。
- (6) 漫长的编辑—编译—调试周期。

Java EE 5 规范的主题就是简化，这一目标已通过改善以下领域的开发体验得到实现：

- (1) 简化业务逻辑开发。
- (2) 简化测试和依赖关系管理。
- (3) 简化 O/R 持久性。
- (4) 增强 Web 服务编程模型。

Java EE 5 中的许多升级都受到商业和开放源代码领域创新技术的影响，例如，Hibernate、Spring、Service Data Object (SDO) 以及其他技术。另外，Java EE 5 还预期对规范的级别进行升级，做一些小幅度的改进。

1.4.2 Java EE的新功能

Java EE 5 是一个功能强大而重要的发行版，是用于企业开发的最完善的版本。显然，它已经采取了一些重要步骤解决了围绕以前 Java 开发的一些问题。EJB 3.0 和 JPA 是功能强大而又易用的技术，而且 JAX-WS 中的增强功能使得 Web 服务的开发比以往更加容易。

为使用 Java EE 5 做准备，本节重点介绍其新规范的一些主要功能，如 EJB 3.0、Java Persistence Architecture (JPA)、Web 服务、JAX-WS 及其他一些内容，然后简单地介绍对 Java EE 6 的一些可能的预期。

1. EJB 3.0

对于 Java EE 5 中的所有技术增强而言，Enterprise JavaBean™ (EJB) 3.0 是最为显著的，已对它的外观进行了很大的更改，明显简化了开发。EJB 3.0 规范已拆分为三个子规范。

(1) EJB 3.0 简化 API: 定义用于编码 EJB 组件（特别是会话 Bean 和消息驱动的 Bean）的新简化的 API。EJB 组件不再要求主接口；J2SE 5.0 标注现在是实现 EJB 3.0 组件的一个主要辅助方法；EJB 3.0 引入了业务接口概念，而非单独的远程和本地接口。

(2) 核心契约和要求: 定义 Bean 和 EJB 容器之间的 EJB 契约。EJB 3.0 规范的更新包括以下内容：容器服务；回调；拦截器；依赖项注入。

(3) Java 持久性体系结构 API: 为持久性定义新实体 Bean 模型。EJB 持久性的规范已发生了显著变化，该规范称为容器管理的持久性 (CMP)，它没有定义映射层，而是由容器映射的，因此它由供应商实现映射。受多个商业和开源产品和技术，如 Hibernate、Java Data Objects (JDO) 和 TopLink 的影响，EJB 3.0 引入了新的持久性样式，即基于 POJO 的持久性。

- ① 在成功模式上建模。
- ② 简化 JDBC 访问模式。
- ③ 可以与 Web 服务集成。
- ④ 不受容器的阻碍，可以在 Java EE 或 Java SE 环境中使用 JPA。
- ⑤ 标准化 O/R 映射元数据。
- ⑥ 支持自顶向下、中间相遇和自底向上的开发。
- ⑦ 支持断接和连接的对象状态，省去了对单独的数据传输对象的需要。

2. JPA API的更新

- (1) 类型：实体和表。
- (2) 实例：Java 对象。
- (3) 属性：Java 属性和列标注。
- (4) 依赖对象：嵌入式 Java 对象。
- (5) 派生属性：瞬态标注。
- (6) 键属性：标注的字段和键类。
- (7) 关系：标注和联合列。

- (8) 约束：标注和数据库。
- (9) 继承：标注——单个表、联合表和按类表。

3. JAX-WS

(1) 从 JAX-RPC 到 JAX-WS 没有发生变化的内容。Java EE 5 还为 Web 服务引入了一个新的编程模型：JAX-WS。在了解不同之处之前，需要知道从 JAX-RPC 到 JAX-WS 哪些内容没有发生变化。

① JAX-WS 仍然支持 SOAP 1.1 over HTTP 1.1，因此互操作性将不会受到影响，仍然可以在网上传递相同的消息。

② JAX-WS 仍然支持 WSDL 1.1，因此所学到的有关该规范的知识仍然有效。新的 WSDL 2.0 规范已经完成，但在 JAX-WS 2.0 相关工作结束时其工作仍在进行中。

(2) 从 JAX-RPC 1.1 到 JAX-WS 2.0 发生的更改。下面大致列出了从 JAX-RPC 1.1 到 JAX-WS 2.0 都发生了哪些更改。

① SOAP 1.2：JAX-RPC 和 JAX-WS 都支持 SOAP 1.1。JAX-WS 还支持 SOAP 1.2。

② XML/HTTP：WSDL 1.1 规范在 HTTP 绑定中定义，这意味着利用此规范可以在不使用 SOAP 的情况下通过 HTTP 发送 XML 消息。JAX-RPC 忽略了 HTTP 绑定，而 JAX-WS 添加了对其的支持。

③ WS-I Basic Profile：JAX-RPC 支持 WS-I Basic Profile (BP)V1.0。JAX-WS 支持 BP 1.1。（WS-I 即 Web 服务互操作性组织。）

④ 新的 Java 功能：JAX-RPC 映射到 Java 1.4。JAX-WS 映射到 Java 5.0。JAX-WS 依赖于 Java EE 5 中的很多新功能（Java EE 5 是 Java EE 1.4 的后续版本，添加了对 JAX-WS 的支持，但仍然支持 JAX-RPC，这可能会对 Web 服务新手造成混淆）。

⑤ 数据映射模型：JAX-RPC 具有自己的映射模型，此模型大约涵盖了所有模式类型中的 90%。那些没有包括的模式被映射到 javax.xml.soap.SOAPElement（JAX-WS 数据映射模型是 JAXB，该模型映射所有 XML 模式）。

⑥ 接口映射模型：JAX-WS 的基本接口映射模型与 JAX-RPC 的区别并不大，不过 JAX-WS 模型使用新的 Java EE 5 功能，并引入了异步功能。

⑦ 动态编程模型：JAX-WS 的动态客户端模式与 JAX-RPC 模式差别很大。根据业界的需要进行了许多更改，其中包括面向消息的功能和动态异步功能。JAX-WS 还添加了动态服务器模型，而 JAX-RPC 则没有此模型。

⑧ 消息传输优化机制（MTOM）：JAX-WS 通过 JAXB 添加了对新附件规范 MTOM 的支持，因此应该能够实现附件互操作性。

⑨ 处理程序模型：从 JAX-RPC 到 JAX-WS 的过程中，处理程序模型发生了很大的变化。JAX-RPC 处理程序依赖于 SAAJ 1.2，而 JAX-WS 处理程序则依赖于新的 SAAJ 1.3 规范。

⑩ JAX-WS 还可以与 EJB 3.0 一起使用来简化编程模型。例如，将 EJB 3.0 POJO 转换为 Web 服务的简单方法。

4. JavaServer Faces

JavaServer Faces (JSF) 到目前为止已有数年的历史，并且受大多数 Java EE 应用服务器支持，如 IBM® WebSphere Application Server。在 Java EE 5 中，JSF 现在是 Java EE 5 规范的一部分。JSF 为 Java EE 应用程序提供了许多优势：

- (1) 丰富、可扩展的 UI 组件。
- (2) 事件驱动。
- (3) 托管组件状态。
- (4) 验证程序。
- (5) 类型转换。
- (6) 外部化导航。
- (7) JavaServer Pages 和 Faces 通用表达语言。

5. 展望Java EE 6

最近专家组对 JSR 316 (Java EE 6) 提出了一些议案, 尽管对其规范定义为时尚早, 但该议案突出了以下几个重要主题。

(1) 可扩展性。通过添加更多扩展点和更多服务提供程序接口, 可以将其他技术简洁高效地插入平台, 从而实现了可增长性。

(2) 概要。概要将根据 JCP 过程的定义参考 Java EE 平台, 并且可能包括 Java EE 平台技术的子集或/和不属于基本 Java EE 平台的其他 JCP 技术。专家组还将定义 Java EE Web 概要的第一个版本, 这是一个用于 Web 应用程序开发的 Java EE 平台的子集。

(3) 技术修剪。Java EE 平台中的一些技术不再具有它们刚引入平台时的相关性。需要有一种方法, 能够认真有序地将这些技术从平台中“修剪”掉, 并让仍使用这些技术的开发人员受到的影响降到最小, 同时使平台能够更健壮地成长。正如该过程定义的那样, 专家组将考虑在将来的 Java EE 平台规范中应将某些技术标记为可被移除。这些可能被移除的技术包括:

- ① EJB CMP, 被 Java Persistence 有效地代替。
- ② JAX-RPC, 被 JAX-WS 有效地代替。

(4) SOA 支持。Java EE 平台已经广泛应用于 SOA 应用程序。随着越来越多的企业认识到 SOA 体系结构的好处, 对该平台的功能和互操作性的需要也相应提高。Java EE 6 需要考虑增加对 Web 服务的支持。尽管基本 Web 服务支持现在是 Java EE 6 平台的一部分, 但此规范将需要这些技术的更新版本, 以便提供更多的 Web 服务支持。服务组件体系结构 (SCA) 定义一些可以在 SOA 环境中由组合应用程序使用的工具。专家组在考虑将所有 SCA 定义的工具都包含在 Java EE 6 平台中是否合适。

(5) 其他增加内容。专家组提议在 Java EE 6 中包括下列新的 JSR:

- ① 用于容器的 JSR—196 Java 身份验证 SPI。
- ② 用于应用服务器的 JSR—236 计时器。
- ③ 用于应用服务器的 JSR—237 工作管理器。
- ④ JSR—299 Web Bean。
- ⑤ JSR—311 JAX—RS: 用于 RESTful Web 服务的 Java API。

预计在以下领域进行进一步更新:

- Enterprise JavaBeans。
- Java Persistence API。
- Servlet。
- JavaServer Faces。

- JAX-WS。
- Java EE Connector API。

具体要包括哪些技术将由专家组根据合作伙伴和客户要求确定，其中一些规范还要考虑到快速发展的 Web 2.0 空间。

1.4.3 Java EE开发环境IDE

Java EE 集成开发环境（Integrated Development Enviroment, IDE）支持完备的应用程序开发周期，包括编写程序代码、测试程序并进行调试、应用程序部署以及效能调整等阶段。目前 Java EE 的集成开发环境 IDE 主要有 JBuilder、Eclipse、NetBeans 等。这里将 JBuilder、Eclipse、NetBeans 三者进行比较。

JBuilder 曾经是 Borland 的产品，在 CodeGear 从 Borland 分离出来之前，一直是 Borland 在指引 JBuilder 前进的方向。作为世界上最成功的 IDE 厂商，Borland 对 JBuilder 的培育还是相当成功的，一度使 JBuilder 成为 Java IDE 的龙头老大。可惜 Borland 不满足于偏安于 IDE 一隅，于是不断地扩大自己的业务范围，逐渐从一个单纯的 IDE 厂商，膨胀到涉及统一建模、需求管理、配置管理、测试、性能调优等诸多领域的庞然大物。IDE 不再是 Borland 的主业，而是变成 ALM 的配角了。ALM 的概念是非常诱人的，一套完整的工具，彼此之间无缝集成，仿佛一条快速运转的生产线，只要将用户的需求从这头放进去，在生产线的另一头，一个合格的软件产品就会如期下线。Borland 一门心思扑在 ALM 上，对 JBuilder 前途的关注，大不如前了，而竞争对手们都没有停下来。

当 Borland 忙于推销自己的 ALM 概念、方法和产品时，Java IDE 的世界也在发生着快速的变化。蓝色巨人 IBM 在与 Borland 的竞争中精疲力竭，它的 Eclipse 远非 JBuilder 的对手，大笔的研发投入，未能给 IBM 带来预期的收益，而狠心放弃，又有些舍不得。正当 IBM 彷徨不定的时候，席卷世界的开源浪潮给 Eclipse 带来了希望。在很多拥有核心技术和知识产权的 IT 巨鳄对开源纷纷持抗拒态度时，IBM 看到了开源的好处，于是非常果断地将“鸡肋”Eclipse 捐给了开源社区。IBM 的这一举动，不但为它赢得了慷慨大方的好名声，而且，也让 Eclipse 获得了新生，在 Java IDE 领域中领先。

Eclipse 最初由 OTI 和 IBM 两家公司的 IDE 产品开发组创建，起始于 1999 年 4 月。IBM 提供了最初的 Eclipse 代码基础，包括 Platform、JDT 和 PDE。目前由 IBM 牵头，围绕着 Eclipse 项目已经发展成为了一个庞大的 Eclipse 联盟，有 150 多家软件公司参与到 Eclipse 项目中，其中包括 Rational Software、Red Hat 及 Sybase 等。

Eclipse 是一个开发源码项目，它其实是 Visual Age for Java 的替代品，其界面跟先前的 Visual Age for Java 差不多，但由于其开放源码，任何人都可以免费得到，并可以在此基础上开发各自的插件，因此越来越受人们关注。近期还有包括 Oracle 在内的许多大公司也纷纷加入了该项目，并宣称 Eclipse 将来能成为可进行任何语言开发的 IDE 集大成者，使用者只需下载各种语言的插件即可。

几乎出于和 IBM 同样的考虑，Sun 也决定将 NetBeans 开源，果然，开源之后的 NetBeans 立刻受到了 Java 开发者的追捧，当 Java 推出新技术或平台新版本的时候，NetBeans 往往是第一个响应者，成为版本更新最迅速的 Java IDE，一直紧随 Sun 的步伐。与此形成鲜明对比的是，Sun 的另一款 Java IDE，面向 Java Web 开发的 Sun Java Studio Creator，由于没有开源，最终难逃被 NetBeans 兼并的命运。NetBeans 是作为一个教学项目于 1996 年启动的，当初的名称并非 NetBeans，而是叫做 Xelfi，1999 年 Sun 介入

NetBeans。Sun Java Studio Creator 要年轻许多，这个后来者是 2001 年才被提上 Sun 的议事日程的，Sun 的意图相当明显：NetBeans 是为 Java 高手打造的精良武器，而 Creator 则是专为 Java 初学者搭建的游乐场。兼并后，将 Creator 作为 NetBeans 的一个插件给 NetBeans 送去了一份大礼，那就是 NetBeans 5.5 中的 Visual Web Pack (VWP)，使 NetBeans 实现了真正意义上的 JSF 可视化开发。NetBeans 的 Visual Web Pack 是 JSF 可视化开发的工具包，具有多项创新的开发技术，正是这些突破常规的手法，造就了 NetBeans 在 JSF 可视化开发领域的先锋地位。

在 Borland 的 JBuilder 称雄 Java IDE 多年后，但最近几年在其他的 Java IDE(如 Eclipse、NetBeans 等)兴起后，从 JBuilder 手中抢走了大量的用户，JBuilder 也在这几年里从鸡腿熬成了鸡肋。而 Borland 在销售它的 IDE 部分失败后，它最新的 JBuilder 2007 一改 Borland 以往的做法，装上 Eclipse 的“芯”又重新上战场了。

当 Borland 全力投入 ALM 业务并且意识到其 IDE 业务已经大受影响时，曾经想过要将 IDE 业务出售。在与几家有能力收购的公司谈过后，收购以失败告终。不管怎样，Borland 最终决定成立一家全资子公司，这就是受命于危难之际在 2006 年 11 月 14 日成立的 CodeGear 公司。CodeGear 接过了 Borland 的全部 IDE 业务，是一家百分之百服务于开发工具市场的公司。JBuilder 2007 是 CodeGear 成立后于 2006 年 11 月 20 日推出的第一个产品，除了 JBuilder 2007 向 Eclipse 的靠拢外，功能上依旧定位于建模和企业级的开发功能，如 EJB 等，虽然加入了对主流 Java 框架的支持，但显然不够彻底，如不支持 JSF 的可视化开发，等等。差不多半年之后，CodeGear 又发布了 JBuilder 2007 的一轮新版本，依然由三个版本组成，分别是面向个人开发者的 JBuilder 2007 Turbo Edition，面向小企业开发者的 JBuilder 2007 Edition 和面向大中企业开发者的 JBuilder 2007 Enterprise Edition。新版本在功能上没有太大的变化，不过有一个值得注意的动向，那就是价格。三个版本中，面向个人开发者的 Turbo Edition 是免费的，而另外两个版本降价幅度分别达到了 38% 和 25%，JBuilder 的忠实追随者或许会放弃 Eclipse 和 NetBeans，重投 JBuilder 的怀抱。

本书使用目前用户最多的 Java EE 可视化集成开发平台——Eclipse。

习 题 1

1. 什么是 Web 应用定义？
2. 简述 Web 应用体系结构（客户/服务器：瘦客户、N 层应用系统）。
3. 简述 Java2 平台的特点，3 个独立的版本的名字及特点。
4. 什么是 Java EE？
5. 简述 Java EE 体系结构。
6. 简述 Java EE 应用程序构成。
7. 在 Java EE 中，部署是什么意思？容器是什么？
8. 简述 Java EE 的由来与发展（从 J2EE 到 Java EE）。
9. Java EE 的新功能有什么？
10. 简述集成开发环境 JBuilder、Eclipse、NetBeans 的比较。

第 2 章 Java EE的可视化集成开发平台—— Eclipse及运行环境

Eclipse 是一个优秀的集成开发环境，同时又是一个可以不断扩展的开放平台。本章主要介绍 Eclipse 的发展历史、现状和重要的功能特性；如何使用 Eclipse 构建一个 Java Web 开发环境；简单介绍如何使用 Eclipse 开发一个插件，并介绍了在后续章节中使用到的重要插件。本章还将介绍 Tomcat Web 服务器、JBoss 应用服务器以及它们与 Eclipse 的集成等，以便读者了解 Java EE 的可视化集成开发平台 Eclipse 及运行环境，为学习后续章节做准备。

2.1 Eclipse概述

Eclipse 是一个非常优秀的集成开发环境，其最初是 IBM 的一个软件产品。Eclipse 项目的初期目标是开发一个替代 IBM VisualAge for Java 的下一代 IDE 开发环境。

IBM 于 2001 年 11 月宣布捐出价值 4000 万美金的开发软件给开源码的 Eclipse 项目，从此 Eclipse 作为一个重要的开放源代码的集成开发工具，受到越来越多的开发者的喜爱，它超越众多的 Java 集成开发工具，截至 2006 年其市场份额超过 60%，成为最有竞争力的 Java 开发工具。前几年流行的 JBuilder 的 2006 新版也建立在 Eclipse 平台之上。

2.1.1 Eclipse的主要特点

虽然大多数用户很乐于将 Eclipse 当做 Java IDE 来使用，但 Eclipse 的目标不仅限于此。Eclipse 还包括插件开发环境（Plug-in Development Enviroment, PDE），这个组件主要针对希望扩展 Eclipse 的软件开发人员，允许他们构建与 Eclipse 环境无缝集成的工具。根据 Eclipse 的体系结构，通过确定插件接口标准，能通过插件扩展到任何语言的开发。Eclipse 作为一个开放源代码的项目，任何人都可以在此基础上开发自己的功能插件。

Eclipse 可比喻为铁匠的工作室，在这里不仅仅可以制作产品，还可以制作制造产品的工具。

Eclipse 是一个平台，其目的是提供一个集成开发工具的必要服务。它拥有一个非常小的运行内核，其他功能可以通过一个或者一系列的插件来实现。当从网上下载了一个 Eclipse SDK 后，就得到了一个编写和调试 Java 程序的 JDT（Eclipse 的 Java 开发工具）和支持 Eclipse 扩展的 PDE。如果想使用 Eclipse 开发 C++ 程序，则可以下载一个 CDT（Eclipse 的 C++ 开发工具）。

插件式的设计使得 Eclipse 具有良好的扩展性，可以在此基础上开发出支持其他语言的开发工具，如 JavaScript、Perl 等。

通过集成大量的插件，Eclipse 的功能可以不断扩展以支持各种不同的应用，Eclipse 的插件可以用于管理多种开发任务，其中包括性能优化、程序调试等，而且还可以集成来自多

个供货商的第三方应用程序开发工具。IBM 公司以 Eclipse 为基础开发出了 Websphere 系列和 Rational 系列的开发工具，BEA 公司在 Eclipse 基础上开发了自己的开发工具 Bean Workshop。另外，一个应用得比较多的开发工具 MyEclipse 也是以 Eclipse 为基础，集成了多种插件形成的。

Eclipse 简化了用于多种操作系统的软件工具的开发过程，可以在多个操作系统上运行，基于 Eclipse 的软件既可以在 Windows 上运行，也可以在 Linux 上执行，省去了开发者有时要把 Windows 应用程序切换到 Linux 的操作，进而简化了整个开发过程。同时，Eclipse 还提供了与每个底层操作系统集成的强大功能。

2.1.2 Eclipse的组成

1. Eclipse的运行内核

Eclipse 是一个开放源代码的通用工具平台，遵循普通公共许可证（Common Public License）进行源代码许可。它专注于为高度集成的工具开发，提供一个全功能的、具有商业品质的公用平台。它最早由 Eclipse 项目、Eclipse 工具项目和 Eclipse 技术项目三个项目组成，最近又新增了 Eclipse Web 工具平台项目。每一个项目由它的子项目组成，并且使用 CPL 版本 1.0 许可协议。

（1）Eclipse 项目，它是 Eclipse 的核心项目、焦点，提供了一个具有丰富特性的开发环境，使开发者有效地建造可以无缝集成到 Eclipse 平台中的工具。它包括如下子项目：Platform（开发作为 Eclipse 的集成开发环境的核心部分）、Java Development Tools（JDT，开发 Java 的开发环境）、Plug-in Development Environment（PDE，开发插件的开发环境）。

（2）Eclipse 工具项目，它为不同的工具建造者提供了焦点，以保证为 Eclipse 平台创建最好的工具。工具项目提供单一的联系点以调和开放源代码工具创建者，从而使得覆盖和重复最小化，并保证共享的最大化和共同组件的创建，促进不同类型工具的无缝互操作。

（3）Eclipse 技术项目，其任务是为开放源代码开发者、研究者、学院和教育者提供新的管道，以参与将来 Eclipse 的演化，分为研究、培育和教育三个项目流：研究项目是指在 Eclipse 相关领域，如编程语言、工具和开发环境进行探索和研究；培育项目是指小型的、未正式结构化的项目，为 Eclipse 软件基础添加新的能力；教育项目聚焦于教育材料的开发教学帮助和课件。

（4）Web 工具平台项目，它可以在 Eclipse 平台项目的基础上建立一套通用框架和服务，以便工具开发者方便地建立可以和符合 Java EE 标准规范的 Web 应用服务器协同工作的 Web 工具集。这个项目的具体目标之一就是尽可能多地支持现有的商用和开放源码的应用服务器。

最常用的 Java 开发功能实际上是由 Eclipse 一个主要的插件 JDT 所提供的，它随 Eclipse SDK 一同发行。JDT 在 Eclipse 平台上实现了 Java 的 IDE，它可以开发任何类型的 Java 程序，包括 GUI 和非 GUI 的程序，甚至包括 Eclipse 的插件。通过 JDT 给 Eclipse 平台引入了 Java 项目、透视图、视图、编辑器、向导、代码合并和重构等概念。

2. Eclipse的结构

Eclipse 除了包括它的运行内核外，还包括工作台、工作区、帮助和团队协作，如图 2-1 所示。

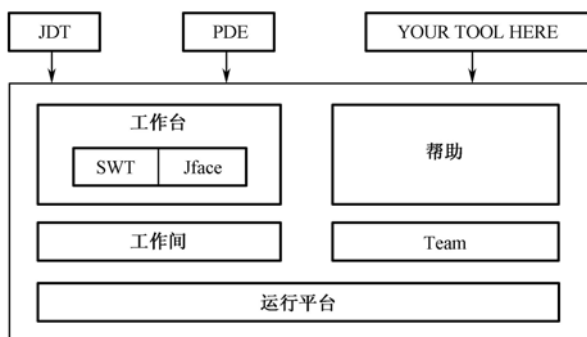


图 2-1 Eclipse 的结构

(1) 工作台是指桌面开发环境。工作台的目标是通过为创建、管理和导航“工作台”资源，提供公共范例来获得无缝的工具集成。通过使用工作台，可以浏览资源，查看和编辑这些资源的内容和特性。每个“工作台”窗口都包含一个或多个透视图。透视图则包含视图和编辑器，并且控制出现在某些菜单栏和工具栏中的内容。在任何给定时间，桌面上可以存在多个“工作台”窗口。

(2) 工作区负责管理使用者的资源，它在顶层被组织成一个或多个项目。每一个项目都是 Eclipse 工作区目录的子目录，每个项目都可以包含多个文件和文件夹。一般来说，每个文件夹都可以看做项目的子文件夹，但是一个文件夹可以连接到文件系统的任何一个目录。

工作区保存了每个资源的部分变化历史，这就使得迅速撤销变化成为可能，同样也可以恢复到先前的保存状态，甚至是几天前的历史数据，这主要取决于用户在历史设置上的配置情况。工作区的这个功能可以将资源丢失的风险降低到最小限度。

工作区还负责管理一个提醒工具，它用来提醒有关工作区资源的变化情况。这个工具还可以为项目标记项目特征，如果是一个 Java 项目，它可以提供代码以配置项目必要的资源。

(3) 联机帮助系统让用户能够浏览、搜索和打印系统文档。文档组织成与书籍类似的信息集。帮助系统还提供了用于按关键字查找所需信息的文本搜索引擎，以及上下文相关帮助。

如果想要在联机帮助中查找特定的信息，可使用“搜索”功能。帮助系统将搜索整个帮助文档集或仅搜索其中某个部分以查找符合指定条件的主题。

如果正在完成某个任务，但遇到了不了解的界面部分，则使用上下文相关帮助。通过单击所处的界面窗口小部件或者使用<Tab>键来使该窗口小部件处于焦点中，然后按<F1>键，弹出信息将提供有关界面的信息，以及指向更多信息的链接。

通过 org.eclipse.help.toc (the table of contents) 扩展点，用户可以添加插件的联机帮助。小工具通常将它们的帮助文档作为代码放在同一插件中，大工具通常有单独的帮助插件。

XML 浏览文件和 HTML 内容文件存储在插件的根目录或子目录中。

(4) 团队协作是指有多人组成团队开发项目，即使只涉及几个开发人员的小项目，也需要对源代码的更改进行严格控制。这就是源代码管理软件的任务。源代码版本控制软件必须支持两个核心功能：

① 提供一种方法，能够协调对源代码的更改，并能集成这些更改。

② 团队所提交工作的历史记录。

当团队成员完成新的工作时，通过将这些更改提交到资源库来共享他们的工作。类似地，当他们希望获得最新可用的工作成果时，就可以根据资源库中的更改，更新自己的本地工作空间。这意味着项目资源库会因团队成员提交新工作成果而经常发生更改。换句话说，资源库应该表示项目的当前状态，即在任何时候，团队成员都能够根据资源库更新自己的工作空间，并确信它们是最新的。

Eclipse 广泛地支持各种代码管理解决方案，提供了对于直接从工作空间进行团队开发操作的支持。这种支持允许开发人员并发地与几个独立的资源库，以及不同版本的代码或项目进行交互。当然，单个工作空间可以同时访问不同类型的资源库。Eclipse 平台并没有提供自己的代码管理解决方案，它总是依靠外部系统，这要归功于它的插件体系结构。Eclipse 平台只对于一个（但也是最流行的一个）源代码管理系统——并发版本控制系统（Concurrent Versions System, CVS）提供内置，通过第三方插件可以支持 SVN、ClearCase 等源代码管理系统。

2.2 Eclipse的安装及开发环境的搭建

搭建 Eclipse 的 Java 集成开发环境一般需要三步：①下载和安装 JDK；②下载并解压缩 Eclipse SDK；③安装其他需要的插件。

2.2.1 下载和安装JDK

Eclipse 本身不包含 Java 的运行环境，要使用 Eclipse 必须先安装 JDK。JDK 既提供了 Java 的运行环境，也提供了简单的 Java 开发环境。JDK 包含了所有编译、运行 Java 程序所需要的工具，其核心 Java API 为 Java 开发者提供了使用 Java 运行环境的功能。

Sun公司和IBM公司都提供了JDK。本书使用Sun公司提供的JDK，可以到下面的地址下载：<http://java.sun.com/javase/downloads/index.jsp>，在下载页面选择适合自己操作系统的JDK版本。

安装时可以选择安装到任意的硬盘驱动器上，如安装到 D: \jdk1.5.0 目录下，如图 2-2 所示。



图 2-2 JDK 的安装

正确安装后，在 jdk 1.5.0 目录下将有 bin、demo、lib、jre 等子目录，如图 2-3 所示。其中，bin 目录保存了 javac、java、appletviewer 等命令文件；demo 目录保存了许多 Java 的例子；lib 目录保存了 Java 类库文件；jre 目录保存了 Java 的运行时环境。

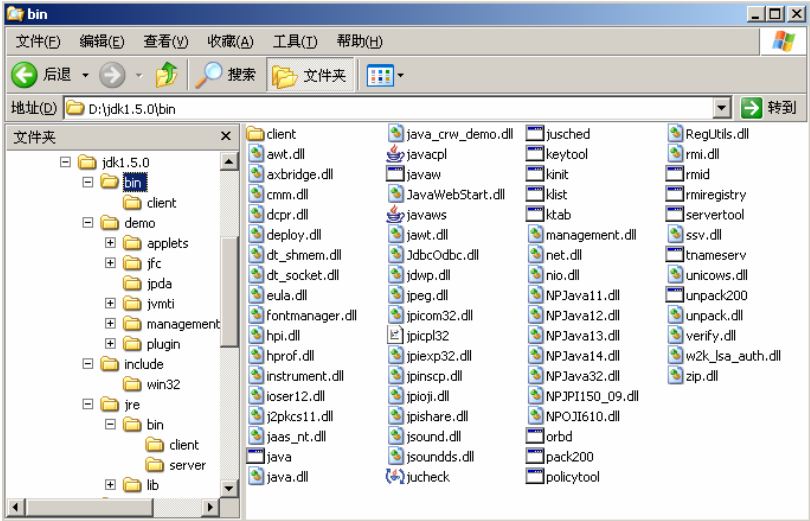


图 2-3 JDK 的目录结构

2.2.2 下载并解压缩Eclipse SDK

Eclipse是一个免费的软件，可以从 www.eclipse.org 下载，在下载页面选择适合自己操作系统的版本。对于一般要求的Java开发者，只下载Eclipse SDK就可以了；对于需要开发Web程序的用户还需要下载WTP工具包，WTP工具包包含了开发Servlet、JSP、XML等语言的开发工具。为了方便用户的使用，Eclipse网站提供了Eclipse SDK和WTP工具包集成的软件包。本书使用包含WTP的集成Eclipse开发环境。

进入 Eclipse 网站，在导航菜单中单击 downloads，在打开页面的左侧导航菜单中选择 By Topic，在新打开页面中按照主题分类显示可以下载的文件，在 Enterprise Development 类别中选择 Web Tools Platform (WTP)，打开新页面后选择一个稳定的版本，在新页面中下载 Windows 版本。

下载后，把它解压缩到硬盘一个目录中，解压后文件的结构如图 2-4 所示。

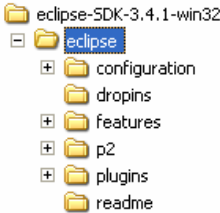


图 2-4 Eclipse 的目录结构

Eclipse 网站还提供了多国语言包插件，可以根据需要下载语言包，把语言包解压缩到和 Eclipse SDK 同一个目录即可。

在 Eclipse 目录下，找到 Eclipse.exe 文件，运行该文件就可以启动 Eclipse 了，可以在桌面上创建一个快捷方式。在第一次启动 Eclipse 时，会提示用户选择个工作间的位置，使用默认值即可。Eclipse 启动后的界面如图 2-5 所示。

使用 Eclipse 开发 Java 程序需要先创建一个项目，然后创建 Class 文件。

(1) 单击“File”→“New”→“Project”，界面如图 2-6 所示。

(2) 在弹出的“New Project”对话框中选择“Java Project”，单击“Next”按钮，如图 2-7 所示。

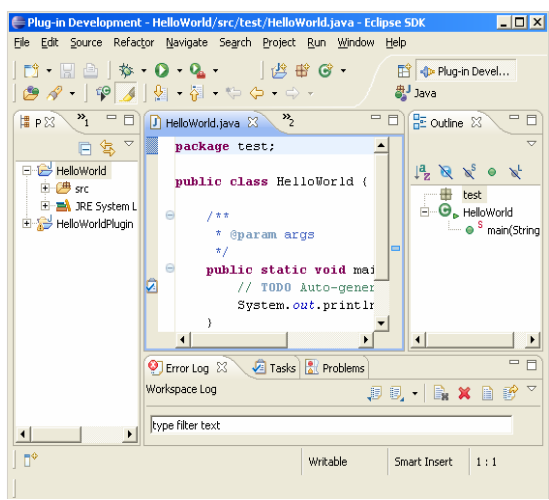


图 2-5 Eclipse 的界面

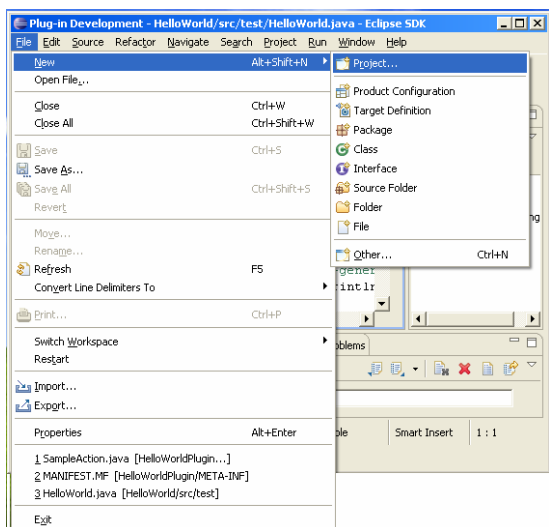


图 2-6 在 Eclipse 中新建 Java 项目

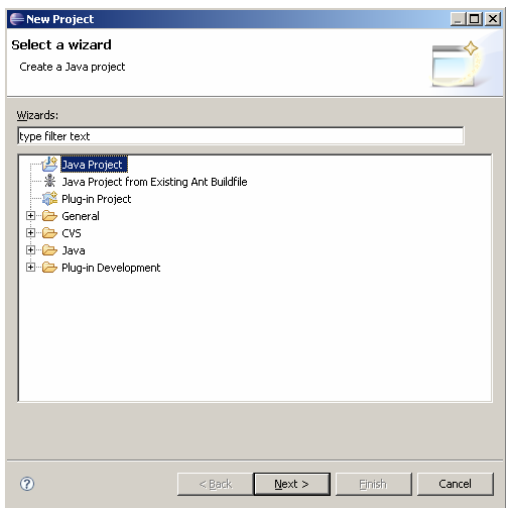


图 2-7 选择创建向导

(3) 在弹出的“New Java Project”对话框中输入项目的名字 HelloWorld，其他设置如图 2-8 所示。输入完成后，单击“Finish”按钮，弹出如图 2-9 所示的窗口。

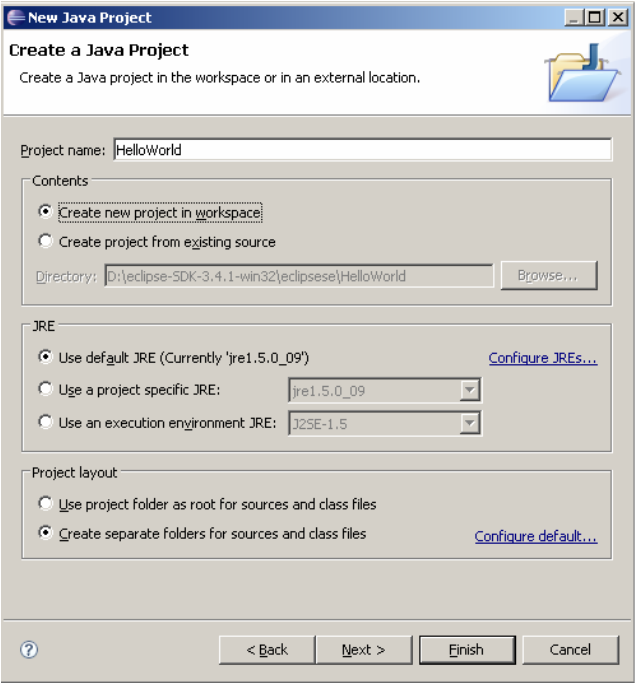


图 2-8 创建一个“HelloWorld”

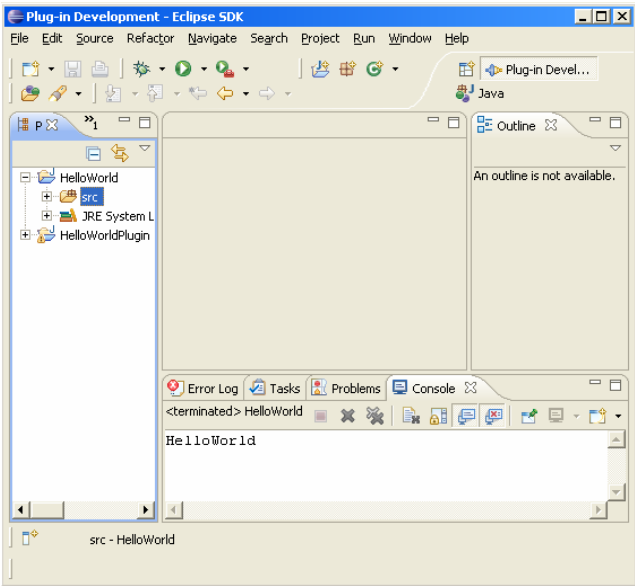


图 2-9 创建 HelloWorld 项目后的窗口

(4) 选择 HelloWorld 项目，单击菜单下面工具栏左边第一个快捷图标，如图 2-10 所示，并单击“Class”选项，打开“New Java Class”对话框，输入包和 Class 的名字，如图 2-11 所示，然后单击“Finish”按钮。

(5) 双击打开创建的 HelloWorld.java 类，在其中输入一行代码，如图 2-12 所示。

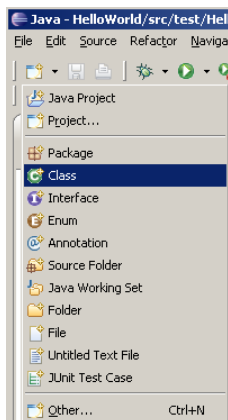


图 2-10 使用快捷图标创建 Class

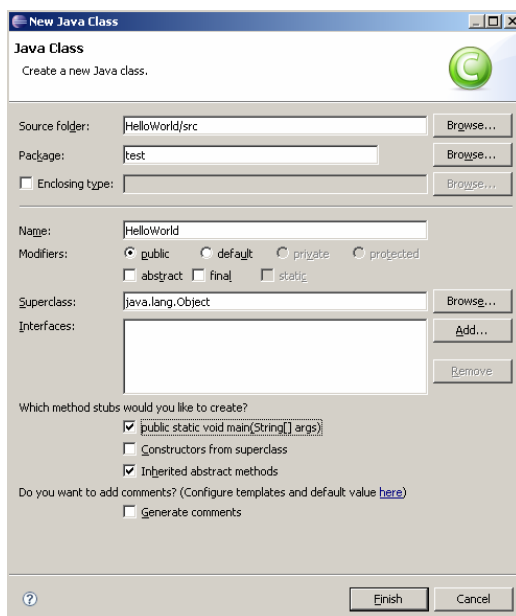


图 2-11 “New Java Class”对话框

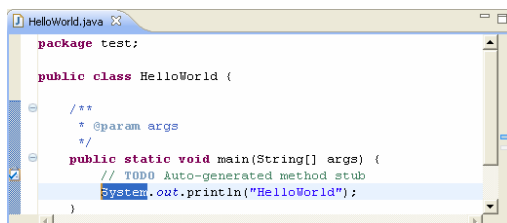


图 2-12 修改 HelloWorld.java

(6) 在 HelloWorld.java 文件编辑器区内，单击鼠标右键，弹出快捷菜单，如图 2-13 所示，选择“Run As”→“Java Application”命令，然后在编辑器下文的“Console”视图会有输出结果，如图 2-14 所示。

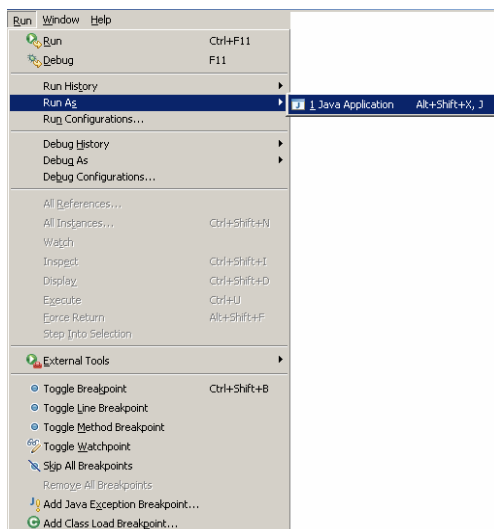


图 2-13 运行 Java 程序

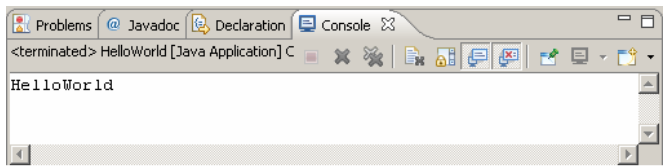


图 2-14 HelloWorld 输出结果

2.2.3 安装Eclipse插件

Eclipse 最有魅力的地方就是它的插件体系结构，它为创建可扩展的集成开发环境提供了一个开放源代码平台。在这样的开放式体系结构中，可让每个插件开发小组可以专注于他们擅长的领域。插件是最小的 Eclipse 平台功能单元，通常简单的工具模块可以写成一个插件，复杂的工具模块可以分割成几个插件，通过插件可以扩展 Eclipse 的功能。

目前互联网上有很多关于 Eclipse 的插件，可以通过 Eclipse 网站或者以下网址查找需要的插件：

<http://www.eclipseplugincentral.com>

<http://www.eclipse-plugins.info/eclipse/index.jsp>

Eclipse 对这些插件是动态载入，动态调用的。所谓动态是指 Eclipse 启动后要真正用到这个插件时，它才会被调入内存。当插件不再被使用时，它就会在适当的时候被清除出内存。因此，即使安装了一大堆插件在 Eclipse 里，也不必担心某些不常用的插件占用内存。

插件的安装方式一般有两种：一种是手工方式，下载插件的压缩包解压到 Eclipse 的 plugins 目录；另外一种给出插件地址自动在线安装。

这里讲解第一种方式，后续章节根据需要讲解其他插件的安装方式。

在后续章节中，将讲解如何在Eclipse中使用Struts框架，为了方便在应用程序的开发中使用Struts框架，需要一个Eclipse插件的支持。使用的插件的名字是StrutsIDE，可以在http://amateras.sourceforge.jp/cgi-bin/fswiki_en/wiki.cgi上下载。

从这个网站上下载两个插件：一个是 EclipseHTMLEditor，另一个是 StrutsIDE。下载的文件名分别是 tk.eclipse.plugin.htmleditor_2.0.6.1 zip 和 tk.eclipse.plugin.struts_2.0.6.zip，分别把这两个文件解压缩到 Eclipse 目录下。如图 2-15 所示，用线框标识出了刚安装的插件。

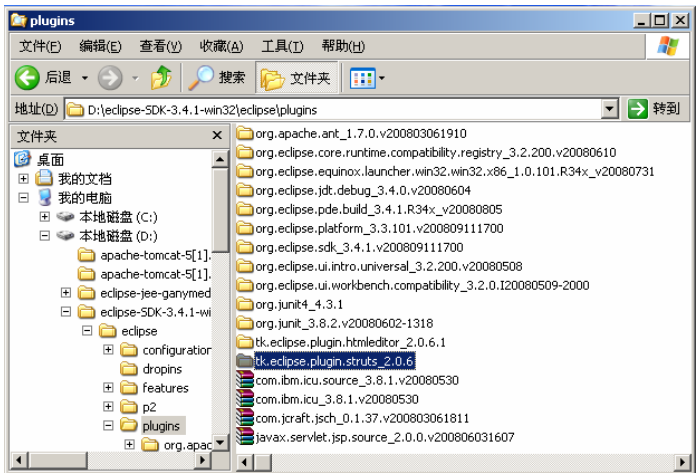


图 2-15 Eclipse 的插件目录

在重新启动时增加“-clean”参数，如图 2-16 所示，在以后启动时取消该参数。

启动 Eclipse 后，单击左边第一个快捷图标，打开对话框，如图 2-17 所示，说明插件安装成功。



图 2-16 为 Eclipse 启动增加参数

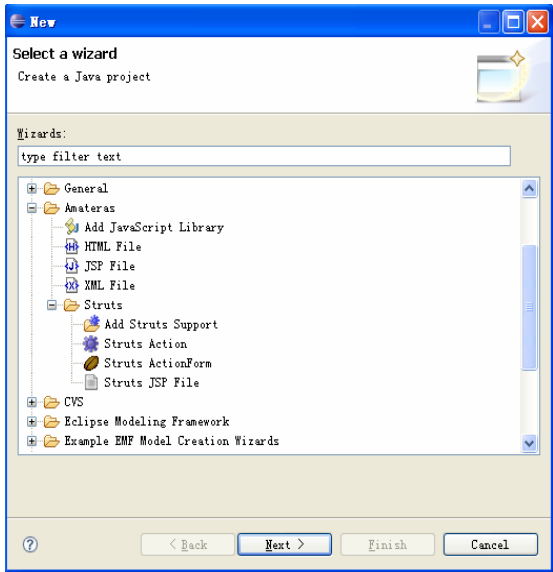


图 2-17 安装插件后的“New”对话框

2.3 Eclipse插件的开发及分类

2.3.1 基于插件的体系结构

Eclipse 平台是 IBM 向开放源码社区捐赠的开发框架，它是一个成熟、精心设计、可扩展的体系结构。Eclipse 的价值是它为创建可扩展的集成开发环境提供了一个开放源代码平台。这个平台允许任何人构建与环境和其他工具的无缝集成工具。

工具与 Eclipse 无缝集成的关键是插件。除了小型的运行时内核之外，Eclipse 中的所有功能部件都是插件。从这个角度来讲，所有功能部件都是以同等的方式创建的。但是，某些插件的作用会更为重要。Workbench 和 Workspace 是 Eclipse 平台的两个必备的插件，它们提供了大多数插件使用的扩展点，如图 2-18 所示。插件需要扩展点才可以插入，这样它才能运行。

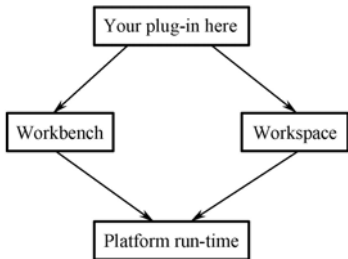


图 2-18 插件扩展点

Workbench 组件包含了一些扩展点，例如，允许用户的插件扩展 Eclipse 用户界面，使这些用户界面带有菜单选择和工具栏按钮；请求不同类型事件的通知；以及创建新视图。Workspace 组件包含了可以让用户与资源（包括项目和文件）交互的扩展点。

当然，其他插件可以扩展的 Eclipse 组件并非只有 Workbench 和 Workspace。还有一个 Debug 组件，可以让用户的插件启动程序与正在运行的程序交互并处理错误，这是构建调试器所必需的。虽然 Debug 组件对于某些类型的应用程序是必需的，但大多数应用程序并不需要它。

还有一个 Team 组件允许 Eclipse 资源与版本控件系统（VCS）交互，但除非用户正在构建 VCS 的 Eclipse 客户机，否则，Team 组件就像 Debug 组件一样，不会扩展或增强它的功能。

此外，还有一个 Help 组件可以让用户提供应用程序的联机文档和与上下文敏感的帮助。帮助文档是专业应用程序必备的部分，但它并不是插件功能的必要部分。

上述每个组件提供的扩展点都记录在 Eclipse Platform Help 中，该帮助位于 Platform Plug-in Developer 指南的参考部分中。

2.3.2 开发HelloWorldPlugin插件

创建插件最简单的方法是使用 Plug-in Development Environment（PDE）。PDE 和 Java Development Tooling（JDT）IDE 是 Eclipse 的标准扩展。PDE 提供了一些向导以帮助创建插件，包括将在这里讲解的“HelloWorld”示例。

（1）从 Eclipse 菜单中打开新建项目对话框，选择“Plug-in Project”，单击“Next”按钮，在对话框中输入项目名称“HelloWorldPlugin”，如图 2-19 所示。单击“Next”按钮，在弹出的对话框中再单击“Next”按钮。

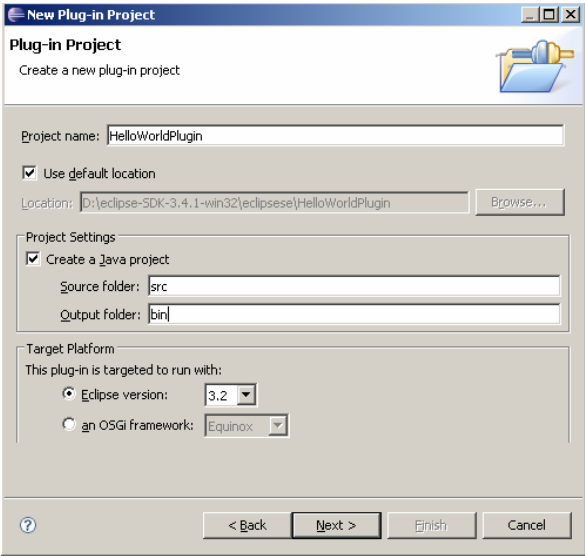


图 2-19 新建插件项目

（2）选择创建插件的模板，这里选择“Hello, World”，如图 2-20 所示。单击“Next”按钮，在对话框中接受默认输入，单击“Finish”按钮完成创建向导，在工作区中会出现一个新的项目“HelloWorldPlugin”，如图 2-21 所示。

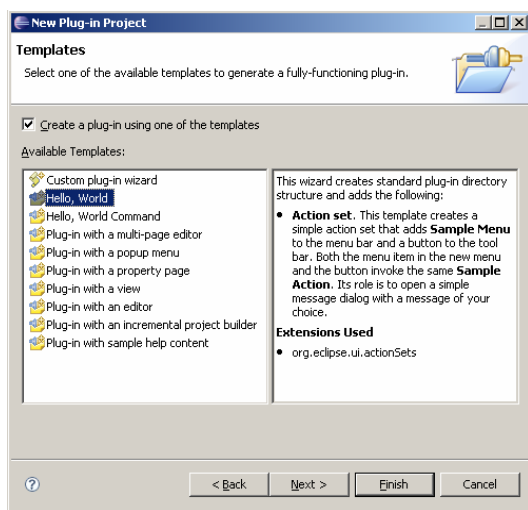


图 2-20 选择插件模板

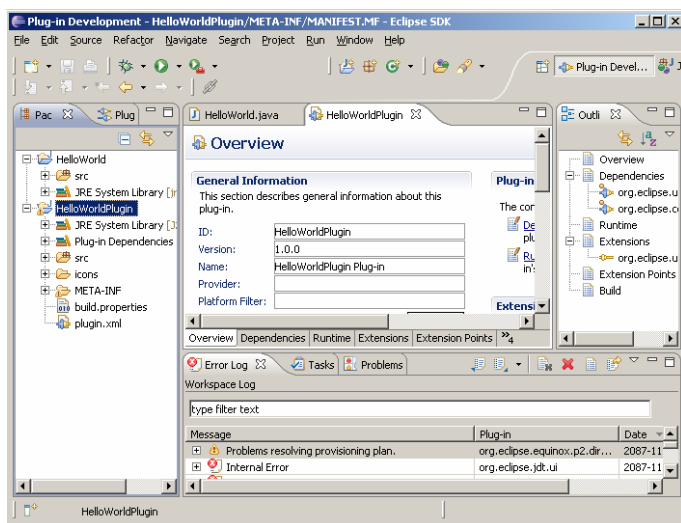


图 2-21 HelloWorldPlugin 插件项目

插件清单文件 `plugin.xml` 包含了 Eclipse 将插件集成到框架所使用的描述信息。默认情况下，当第一次创建插件时，会在清单编辑器区域中打开 `plugin.xml`。编辑器底部的选项卡让用户可以选择关于插件的不同信息集合，首先看到的是“Overview”选项卡的内容。选择“Plugin.xml”选项卡可以查看 `plugin.xml` 文件的完整源代码。

“Overview”选项卡中主要包括：基本信息（General Information）、执行环境（Execution Environment）、插件内容（Plug-in Content）、扩展（Extensions）、测试（Testing）和导出（Exporting）。

“基本信息”包括插件的标识、名称和版本等，插件的标识一般就是插件项目的名称。这些信息可能在 `plugin.xml` 文件中看不到，而是位于 `manifest.mf` 文件中。Eclipse 为了避免直接修改 `plugin.xml` 文件引发错误，隐藏了某些信息，提供了可视化的修改界面。

“执行环境”设置执行这个插件所需要的最小运行环境，包括 Java 运行时环境等。

“插件内容”包括编译和运行该插件而依赖的其他插件和类库。切换到“dependencies”选项卡可以看到该插件需要另外两个插件：`org.eclipse.ui` 和 `org.eclipse.core.runtime`。第一个

插件是工作台插件，HelloWorldPlugin 插件就是在此插件上做出扩展，新建一个 action；第二个插件是运行时环境。

“扩展”包括该插件所做的扩展使用的扩展点。列出了我们在扩展点 org.eclipse.ui.actionSets 上做出的扩展是一个 Sample Action，相当于工具栏上的一个动作图标。

“测试”包括如何运行和调试插件，单击“Launch an Eclipse Application”就可以运行开发的插件。

“导出”部分提供了一些向导，帮助把插件打包和导出到一个目录或者压缩文件中。

(3) 向导产生了一个 SampleAction 类，修改 run() 方法的代码，如图 2-22 所示。

(4) 运行插件。在“Overview”选项卡的“Tesing”区域，单击 Launch an Eclipse Application 就可以运行用户开发的插件，Eclipse 会启动一个新的工作台，在工作台的快捷图标区域会增加一个刚开发的插件的图标。

单击该图标就会运行这个插件，显示一个对话框，如图 2-23 所示。

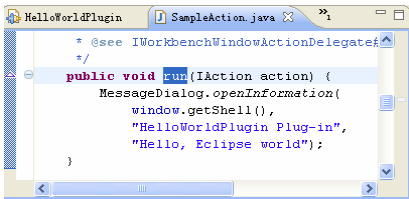


图 2-22 SampleAction 的 run() 方法

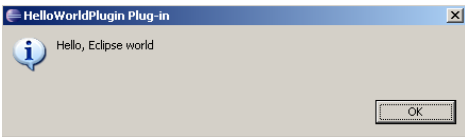


图 2-23 插件运行图

(5) 打包插件。Eclipse 在启动时会查看其插件目录来确定要装入哪些插件。要安装插件，需要在插件目录中创建一个子目录，并将程序文件和清单文件复制在那里。建议目录名称能表示插件的标识，并且后面附带下画线的版本号，但是这种做法不是必需的。假设 Eclipse 安装在 E:\eclipse 中，要创建一个目录：

E:\eclipse\plugins\helloworldplugin_1.0.0

把导出的文件复制到这个目录中即可。

打开插件项目，在插件的“Overview”选项卡的“Exporting”区域中选择“Exporting Wizard”，如图 2-24 所示。

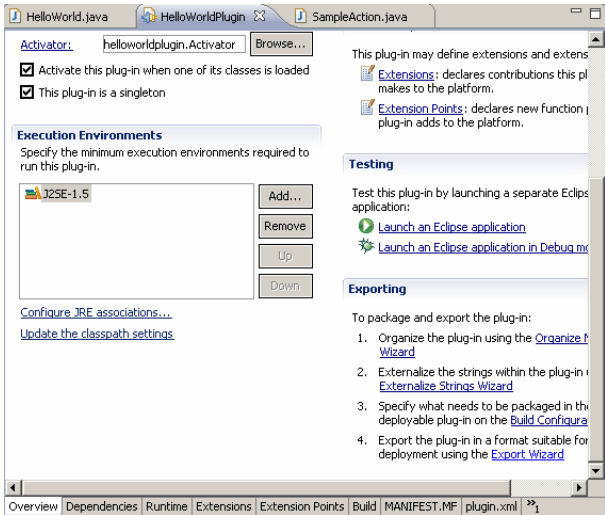


图 2-24 选择插件导出向导

在“Export”对话框中，输入目的导出路径，如图 2-25 所示。

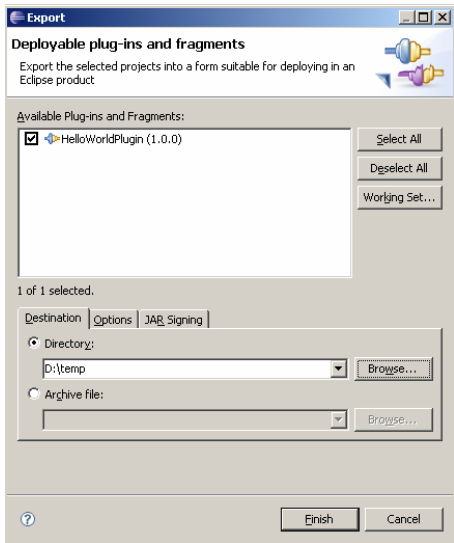


图 2-25 插件导出向导

导出结果放在 HelloWorldPlugin_1.0.0.jar 文件中，包含了开发的插件的所有内容，然后把 HelloWorldPlugin_1.0.0.jar 复制到 Eclipse 的 Plugins 目录中可使用。

2.3.3 Eclipse 插件的分类

目前在 <http://www.eclipseplugincentral.com/> 上收录了 707 个插件，在 <http://eclipse-plugins.2y.net/eclipse/plugins.jsp> 上收录了 1365 个插件，并对插件进行了分类。本节对一些主要的插件进行介绍。

1. Web

- (1) EclipseHTMLEditor: 在 Eclipse 中可视化设计 HTML 和 JSP 等。
- (2) GWT Designer: 一个可以使用 Google Web Toolkit 进行可视化设计的工具，可以自动生成 Java 代码。
- (3) Http Headers: 提供一个简单的图形界面可以获取 Http 头信息。
- (4) Struts Explorer: 方便查看 Struts 的配置文件，而不能进行编辑。
- (5) CSS Editor: 可视化编辑样式表文件。
- (6) Struts Console: 一个独立执行的 Java 程序，可以开发和管理 Struts 程序。
- (7) Easy Struts: 支持用户开发 Struts 程序，但是不支持 Eclipse3。
- (8) Struts: 帮助用户管理大型 Struts 程序的复杂性。
- (9) BlastJ for Struts: 帮助 Java 开发者产生 Struts 应用程序的配置文件和代码。
- (10) MyEclipse: 集成了 Struts、Hibernate 和 Spring 等众多插件的商业插件。

2. XML

- (1) Altova XMLSpy: 把 XMLSpy 和 Eclipse 集成，提供完整的 XML 开发功能。
- (2) XMLBuddy: 支持内容辅助、语法颜色、内容验证和文档格式化等，推荐使用。
- (3) Oribe XML: 支持 XML 开发，支持树形结构显示 XML，内容辅助，DTD 和

Schema 之间的转换等。

(4) XMLAuthor: 提供用户所期望的编辑 XML 的功能, 如语法高亮度显示、错误报告、大纲和导航支持、内容辅助和自动完成、元素折叠、内容格式化、DTD 和 Schema 验证等。

3. Application Management

(1) Exadel Studio Pro: 提供高级企业 Web 程序开发环境, 功能很强, 商业软件, 其中的标准版是免费的, 功能受到限制。

(2) Eclipse Explorer: 可以把 Eclipse 中的文件夹或者文件在资源管理器中定位。

(3) BEA Workshop Studio: 基于 Eclipse 提供全方位的 Web 及企业家应用程序开发支持。

4. Application Server

(1) Merve Tomcat Launcher Plugin: 在 Eclipse 中启动 Tomcat。

(2) Sysdeo Tomcat Launcher: 在 Eclipse 中运行和调试 Tomcat 程序。

2.4 Web服务器和应用服务器

2.4.1 Web服务器和应用服务器简介

1. Web服务器

Web 服务器 (Web Server) 也叫 WWW 服务器, 是指驻留于因特网上某种类型计算机上的程序 (服务器)。当 Web 浏览器 (客户端) 连到服务器上并请求文件时, 服务器将处理该请求, 并将文件发送到该浏览器上, 附带的信息会告诉浏览器如何查看该文件 (即文件类型)。服务器使用 HTTP (超文本传输协议) 进行信息交流, 它有以下三方面的特点:

(1) 应用层使用 HTTP 协议。

(2) HTML 文档格式。

(3) 浏览器采用统一资源定位器 (URL) 来请求资源。

Web 服务器可以解析 HTTP 协议。当 Web 服务器接收到一个 HTTP 请求 (Request) 时, 会返回一个 HTTP 响应 (Response)。例如, 送回一个 HTML 页面。为了处理一个请求, Web 服务器可以响应一个静态页面或图片进行页面跳转 (Redirect), 或者把动态响应 (Dynamic Response) 的产生委托 (Delegate) 给其他程序。例如, CGI 脚本、JSP 脚本、Servlets、ASP 脚本、服务器端 JavaScript 或者一些其他的服务器端 (Server-side) 技术。无论它们的目的如何, 这些服务器端的程序通常产生一个 HTML 的响应来供浏览器浏览。

Web 服务器的代理模型非常简单。当一个请求被送到 Web 服务器时, 它只单纯地把请求传递给可以很好地处理请求的程序 (即服务器端脚本)。Web 服务器仅仅提供一个可以执行服务器端程序和返回 (程序所产生的) 响应的环境, 而不会超出职能范围。服务器端程序通常具有事务处理 (Transaction Processing)、数据库连接 (Database Connectivity) 和消息 (Messaging) 等功能。

常见的 Java 技术的 Web 服务器有 Tomcat、Resin、Jetty、Orion 等。

2. 应用程序服务器

目前在 Internet/Intranet/Extranet 环境中, 企业级应用系统大多采用三层或多层应用模

式。为了方便开发、部署、运行和管理基于多层结构的应用，需要以网络和分布式计算的底层技术为基础，构建一个完整的应用框架，提供相应的支撑平台作为多层应用的基础设施，这一支撑平台的关键就是位于中间层的应用服务器。应用服务器是一个创建、部署、运行、集成和维护多层分布式企业级应用的平台。如果应用服务器与 Web 服务器相结合，或者包含了 Web 服务器的功能，则称之为 Web 应用服务器。

应用服务器（Application Server）是具有一整套集成分布式计算能力的软件服务器产品。它管理客户请求，为业务逻辑提供宿主环境、连接数据、事务处理、目录等后端计算资源。从历史的角度来看，Web 应用服务器是从各种中间件产品和技术中蜕化而来的。

更简单地说，Web 服务器+EJB 容器就是 Java EE 应用服务器。

在 Web 技术出现的早期，HTTP 服务器主要用于向客户机提供静态 HTML 主页。随着 Internet 逐步走向成熟，CGI/Perl 脚本语言和 Coldfusion 等技术又为 Web 服务器提供了业务逻辑和数据库访问能力。之后，这些技术的局限性、基于 Web 事务处理的高要求，以及全球电子商务稳定增长等几个因素，进一步刺激了一些传统中间件供应商充当急先锋，开始提供基于 Web 的解决方案。

应用服务器把数据库信息（通常来源于一个数据服务器）与终端用户或者客户端程序（常常在 Web 浏览器里运行）连接在一起。在这个连接中很有必要设置一个中间件，因为这个中间件可以减小客户端程序的大小和复杂性，且可缓存，更好地控制数据流，以提供更好的性能，来为数据通信和用户通信提供安全保障。

在企业应用中，应用服务器可以提供以下服务：提高企业应用开发的有效性，保障业务逻辑和组件的重用性；提高企业应用的性能，例如，高运行性能和响应时间、可伸缩性、可靠性等；使企业应用更易于监控和管理，降低系统维护和升级成本。由于应用服务器的重要作用 and 关键地位，它已经成为当今业界的一个热点。

近年在应用服务器市场上最具意义的进展，就是 Java EE 的出现。Java EE 是 Sun 公司提出的开发、部署、运行和管理基于 Java 分布式应用的标准平台。它以 Java 2 平台标准版（J2SE）为基础，继承了标准版的许多优点，还提供了对 EJB、Java Servlet、JSP 等技术的全面支持。Java EE 使用 EJB Server 作为商业组件的部署环境，在 EJB Server 中提供了分布式计算环境中组件需要的服务，例如，组件生命周期的管理、数据库连接的管理、分布式事务的支持、组件的命名服务，等等。Java EE 用于实现应用服务器有其优势，它可以利用 Java 语言自身具有的跨平台性、可移植性、面向对象、内存管理等方面的性能，为应用服务器的实现提供一个完整的底层框架。Java EE 中定义的各种服务，包括 JSP 和 Servlet 容器、EJB 容器、JDBC、JNDI（名字目录服务）、JTS/JTA（事务服务）、JMS（消息服务）等，也分别为应用服务器提供了各种支持。

在 Java EE 应用服务器之上开发的代码，最大的特点是具有非常强的可移植性。例如，在 BEA 公司 Weblogic 服务器上开发的 Servlet 可以部署到 IBM 公司的 Websphere 服务器上，而不需要经过任何代码级修改。

为了确保在不同供应商服务器产品上开发的应用组织之间的兼容性，Sun 公司发布了 Java EE 许可证计划和 Java EE 兼容性测试包（CTS）。获得 Java EE 技术许可并通过 CTS 测试的供应商可以称其产品为“通过鉴定的 Java EE”。

目前，基于 Java EE 的应用服务器主要有 IBM Websphere、BEA WebLogic、JBoss Application Server、Geronimo、Oracle AS、Sun iPlanet、SilverStream eXtend 等。Resin 也可

以作为应用服务器来使用。应用服务器还可以基于 Microsoft.NET 解决方案和其他技术，本书不再赘述。

2.4.2 Tomcat Web服务器

Tomcat 和 Resin 是目前最流行的 Java Web 服务器，下面将介绍 Tomcat 服务器的原理、结构和简单的使用。

1. Tomcat的概念

Tomcat 是 Apache Jakarta 软件组织的一个子项目，Tomcat 是一个 JSP/Servlet 容器，它是在 Sun 公司的 JSWDK（Java Server Web Development Kit）基础上发展起来的一个 JSP 和 Servlet 规范的标准实现，使用 Tomcat 可以体验 JSP 和 Servlet 的最新规范。经过多年的发展，Tomcat 不仅是 JSP 和 Servlet 规范的标准实现，而且具备了商业 Java Servlet 容器的特性，并被一些企业用于商业用途。

Tomcat 作为 Servlet 容器，有 3 种工作模式：独立的 Servlet 容器、进程内的 Servlet 容器和进程外的 Servlet 容器。

(1) Tomcat 作为独立的 Servlet 容器时，它是内置在 Web 服务器中的部分，是指使用基于 Java 的 Web 服务器的情形。例如，Servlet 容器是 Java Web Server 的一部分。独立的 Servlet 容器是 Tomcat 的默认模式。然而，大多数的 Web 服务器并非基于 Java。所以 Tomcat 又发展了其他两种工作模式与非基于 Java 的 Web 服务器结合。

(2) Tomcat 作为进程内的 Servlet 容器时，Servlet 容器作为 Web 服务器的插件与 Java 容器相结合。Web 服务器插件在内部地址空间打开一个 JVM（Java Virtual Machine）使 Java 容器得以在内部运行。若有某个需要调用 Servlet 的请求，插件将取得对此请求的控制并将它传递（使用 JNE）给 Java 容器。进程内的容器对于多线程、单进程的服务器非常适合，并且提供了很好的运行速度，只是伸缩性有所不足。

(3) Tomcat 作为进程外的 Servlet 容器时，Servlet 容器运行于 Web 服务器之外的地址空间，并且作为 Web 服务器的插件与 Java 容器相结合。Web 服务器插件和 Java 容器 JVM 使用 IPC 机制（通常是 TCP/IP）进行通信。当一个调用 Servlet 的请求到达时，插件将取得对此请求的控制并将其传递给 Java 容器，进程外容器的反应时间或进程外容器引擎不如进程内容器，但进程外容器引擎在许多其他方面如伸缩性，稳定性等性能更好。

Tomcat 既可作为独立的容器（主要用于开发与调试），又可作为对现有服务器的扩展（当前支持 Apache、IIS 和 Netscape 服务等）。所以在配置 Tomcat 时，必须决定如何应用它，如果选择第 2 或第 3 模式，还需要安装一个 Web 服务器接口。

2. Tomcat的组织结构

Tomcat 是一个基于组件的服务器，它的构成组件都是可配置的，其中最外层的组件是 Catalina Servlet 容器，其他的组件按照一定的格式要求配置在这个顶层容器中。Tomcat 的各个组件是在<TOMCAT_HOME>\conf\server.xml 文件中配置的。Tomcat 服务器默认情况下对各种组件都有默认的实现，下面通过分析 server.xml 文件来理解 Tomcat 的各个组件是如何组织的。

1) Tomcat 的组件

server.xml 文件的基本组成结构如下：

<Server>	顶层类元素：可以包括多个 Service
<Service>	顶层类元素：可包含一个 Engine，多个 Connector
<Connector/>	连接器类元素：代表通信接口
<Engine>	容器类元素：为特定的 Service 组件处理客户请求，要包含多个 Host
<Host>	容器类元素：为特定的虚拟主机组件处理客户请求，可包含多个 Context
<Context>	容器类元素：为特定的 Web 应用处理所有客户请求
</Context>	
</Host>	
</Engine>	
</Service>	
</server>	

以上的代码就是 server.xml 文件的基本组成结构，一个元素代表一个组件。其中，Server 组件对应<Server>元素，它是配置文件的最顶层元素，代表一个服务器。一个配置文件中只能有一个<Server>元素。

Service 元素由一个或者多个 Connector 组成，另外还包括一个 Engine，负责处理所有 Connector 所获得的客户请求。

一个 Connector 将在某个指定端口上侦听客户请求，并将获得的请求交给 Engine 来处理，从 Engine 处获得回应并返回客户。Tomcat 有两个典型的 Connector，一个直接侦听来自 Browser 的 HTTP 请求，另一个侦听来自其他 WebServer 的请求。Coyote HTTP/1.1 Connector 在端口 8080 处侦听来自客户 Browser 的 HTTP 请求，Coyote JK2 Connector 在端口 8009 处侦听来自其他 Web Server（Apache）的 Servlet/JSP 代理请求。

Engine 元素下可以配置多个虚拟主机（Virtual Host），每个虚拟主机都有一个域名。当 Engine 获得一个请求时，它把该请求匹配到某个 Host 上，然后把该请求交给该 Host 来处理。Engine 有一个默认虚拟主机，当请求无法匹配到任何一个 Host 上的时候，将交给该默认 Host 来处理。

Host 代表一个虚拟主机，每个虚拟主机和某个网络域名（Domain Name）相匹配。每个虚拟主机下都可以部署（Deploy）一个或者多个 Web App，每个 Web App 对应于一个 Context。

一个 Context 对应于一个 Web Application，一个 Web Application 由一个或者多个 Servlet 组成。Context 在创建的时候将根据配置文件 \$CATALINA_HOME/conf/web.xml 和 \$WEBAPP_HOME/WEB-INF/web.xml 载入 Servlet 类。当 Context 获得请求时，将在自己的映射表（mapping table）中寻找相匹配的 Servlet 类。如果找到，则执行该类，获得请求的回应并返回。

在 Tomcat 中只有 3 个组件是可以处理客户请求并生成响应的。这 3 个组件分别是 Engine、Host 和 Context 组件，分别代表了不同的服务范围，通过嵌套关系可以知道 3 个组件的范围有如下的关系：Engine>Host>Context。

Engine 组件下可以包含多个 Host 组件，它为特定的 Service 组件处理所有客户请求。

一个 Host 组件代表一个虚拟主机，一个虚拟主机中可以包含多个 Web 应用（Context 组件）。而 Context 组件代表一个 Web 应用，如图 2-26 所示。

在 Sun 的 Java Servlet 规范中，对 Java Web 应用的定义是：Java Web 应用是由一些 Servlet、HTML 页面、Java 类、JSP 页面和一些其他的资源构成的。它可以在各种实现了 Servlet 规范的各种厂商的 Web 应用容器中运行。Tomcat 就是这样一个实现了 Servlet 规范的

Servlet/JSP 容器。

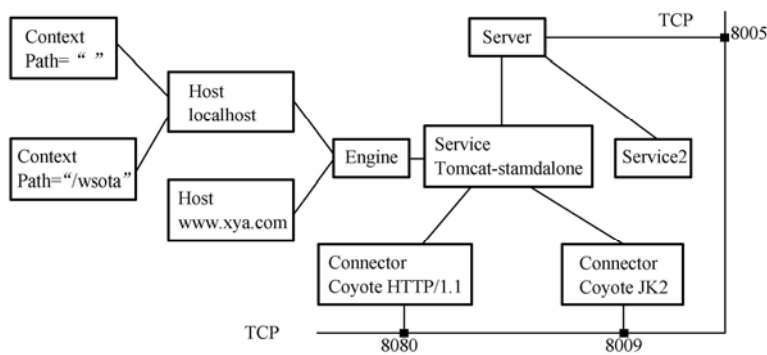


图 2-26 Tomcat 中组件示意图

一个 Java Web 应用在 Tomcat 中与一个 Context 元素对应，也就是说，一个 Context 元素定义了一个 Java Web 应用，它们是一一对应的关系。

2) 配置文件 server.xml 和 web.xml 的内容

server.xml 配置文件的说明见表 2-1。

表 2-1 server.xml 配置文件

元 素 名	属 性	说 明
Server	port	指定一个端口，这个端口负责监听关闭 Tomcat 的请求
	shutdown	指向端口发送的命令字符串
Service	name	指定 service 的名字
	port	指定服务器端要创建的端口号，并在这个端口监听来自客户端的请求
	minProcessors	服务器启动时最小创建的处理请求的线程数
	maxProcessors	最大可以创建的处理请求的线程数
	enableLookups	如果为 true，则可以通过调用 request.getRemoteHost() 进行 DNS 查询来得到远程客户端的实际主机名；若为 false，则不进行 DNS 查询，而是返回其 IP 地址
	redirectPort	指定服务器正在处理 http 请求收到一个 SSL 传输请求后重定向的端口号
Connector（表示客户端和 Service 之间的连接）	acceptCount	当所有可以使用的处理请求的线程数都被使用时，可以放到处理队列中的请求数，超过这个线程数的请求将不予处理
	connectionTimeout	指定超时的时间数（以 ms 为单位）
	defaultHost	指定默认的处理请求的主机名，它至少与其中的一个 host 元素的 name 属性值是一样的
	docBase	应用程序的路径或者是 WAR 文件存放的路径
	path	表示此 Web 应用程序的 url 的前缀，这样请求的 url 为 WAR http://localhost:8080/path/* ** *
	reloadable	该属性非常重要，如果为 true，则 Tomcat 会自动检测应用程序的 /WEB-INF/lib 和 /WEB-INF/classes 目录的变化，自动装载新的应用程序，可以在不重启 Tomcat 的情况下改变应用程序
Context（表示一个 Web 应用程序，通常为文件，关于 WAR 的具体信息见 Servlet 规范）	name	指定主机名
	appBase	应用程序基本目录，即存放应用程序的目录
	unpackWARs	如果为 true，则 Tomcat 会自动将 WAR 文件解压，否则不解压，直接从 WAR 文件中运行应用程序

元素名	属性	说明
Logger（表示日志调试和错误信息）	className	指定 logger 使用的类名，此类必须实现 org.apache.catalina.Logger 接口
	prefix	指定 log 文件的前缀
	suffix	指定 log 文件的后缀
	timestamp	如果为 true，则 log 文件名中要加入时间，如 localhost_log.2006-12- 01.txt
Realm（表示存放用户名、密码及 role 的数据库）	className	指定 Realm 使用的类名，此类必须实现 org.apache.catalina.Realm 接口
Valve（功能与 Logger 差不多，其 prefix、suffix 属性的解释和 Logger 中的一样）	className	指定 Valve 使用的类名，如用 org.apache.catalina.valves.AccessLog Valve 类可以记录应用程序的访问信息
	directory	指定 log 文件存放的位置
	pattern	有两个值，common 方式记录远程主机名或 IP 地址、用户名、日期、 第一行请求的字符串、HTTP 响应代码、发送的字节数；combined 方式比 common 方式记录的值更多

Context 的部署配置文件 web.xml 的说明如下：

一个 Context 对应于一个 Web App，每个 Web App 由一个或者多个 Servlet 组成。当一个 Web App 被初始化时，它将用自己的 ClassLoader 对象载入部署配置文件 web.xml 中定义的每个 Servlet 类。它首先载入在\$CATALINA_HOME/conf/web.xml 中部署的 Servlet 类，然后载入在自己的 Web App 根目录下的 WEB-INF/web.xml 中部署的 Servlet 类。web.xml 文件有两部分：Servlet 类定义和 Servlet 映射定义，每个被载入的 Servlet 类都有一个名字，且被填入该 Context 的映射表（mapping table）中，与某种 URL PATTERN 对应。当该 Context 获得请求时，系统将查询映射表，找到被请示的 Servlet，并执行以获得请求回应。所有 Context 共享的 web.xml 文件，在其中定义的 Servlet 被所有的 Web App 载入。

web.xml 的形式如下：

```
<? xml version = "1.0" encoding = "ISO-8859-1"?>
<! DOCTYPE web-app
PUBLIC "-//Sun Microsystems,Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>FormDeal</servlet-name>
    <servlet-class>FormDeal</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>FormDeal</servlet-name>
    <url-pattern>/FormDeal</url-pattern >
  </servlet-mapping>
  <servlet>
    <servlet-name>UseBeanServlet</servlet-name>
    <servlet-class>UseBeanServlet</servlet-class>
  </servlet>
  <servlet-mapping>
```

```
<servlet-name>UseBeanServlet</servlet-name>
<url-pattern>/UseBeanServlet</url-pattern>
</servlet-mapping>
</web-app>
```

web.xml 中还有其他一些元素，这里就不再赘述了。

3) 实例讲解

下面讲述 Tomcat Server 处理一个 HTTP 请求的过程。

假设来自客户的请求为：

http://localhost:8080/lizhx /lizhx_index.jsp

请求过程如下：

- (1) 请求被发送到本机端口 8080，被在那里侦听的 Coyote HTTP/1.1 Connector 获得。
- (2) Connector 把该请求交给它所在的 Service 的 Engine 来处理，并等待 Engine 的回应。
- (3) Engine 获得请求 localhost/lizhx/lizhx_index.jsp，匹配所有虚拟主机 Host。
- (4) Engine 匹配到名为 localhost 的 Host（即使匹配不到也把请求交给该 Host 处理，因为该 Host 被定义为该 Engine 的默认主机）。
- (5) 名为 localhost 的 Host 获得请求 /lizhx/lizhx_index.jsp，匹配它所拥有的所有 Context。
- (6) Host 匹配到路径为 /lizhx 的 Context（如果匹配不到就把该请求交给路径名为 “ ” 的 Context 去处理）。
- (7) path = “/lizhx” 的 Context 获得请求 /lizhx_index.jsp，在它的 mapping table 中寻找对应的 Servlet。
- (8) Context 匹配到 URL PATTERN 为 *.jsp 的 Servlet，对应于 JspServlet 类。
- (9) 构造 HttpServletRequest 对象和 HttpServletResponse 对象，作为参数调用 JspServlet 的 doGet 或 doPost 方法。
- (10) Context 把执行完了之后的 HttpServletResponse 对象返回给 Host。
- (11) Host 把 HttpServletResponse 对象返回给 Engine。
- (12) Engine 把 HttpServletResponse 对象返回给 Connector。
- (13) Connector 把 HttpServletResponse 对象返回给客户 Browser。

3. Tomcat的安装配置

Tomcat 是基于 Java 的一个 Servlet 容器，它的运行离不开 JDK 的支持。所以，要首先安装 JDK，然后才能正确安装 Tomcat。

1) 安装 Java Development Kit (JDK)

Sun 公司为所有的 Java 程序员提供了一套免费的 Java 开发和运行环境。本书将使用 JDK 5.0 版，可以通过 IE 或 Netscape 浏览器浏览网址：<http://java.sun.com/j2se>，根据提示下载支持 Microsoft Windows 操作系统的 jdk-1_5_0_06-windows-i586-p.exe 到本地硬盘。

安装的时候可以选择安装到任意的硬盘驱动器上。正确安装后，在 JDK 目录下有 bin、demo、lib、jre 等子目录，其中 bin 目录保存了 javac、java、appletviewer 等命令文件；demo 目录保存了许多 Java 的例子；lib 目录保存了 Java 的类库文件；jre 目录保存的是 Java 的运行时环境 (JRE)。

下面设置 JDK 的环境变量。设置环境变量的目的是为了能够正常使用所安装的 JDK 开

发包。通常，我们需要设置 3 个环境变量：JAVA_HOME、PATH 和 CLASSPATH。它们的设置情况见表 2-2。

环境变量名	环境变量值
JAVA_HOME	d:\jdk1.5.0（其内容应根据 JDK 安装目录变化）
PATH	d:\jdk1.5.0\bin;%path%（其内容应根据 JDK 安装目录变化）
CLASSPATH	d:\jdk1.5.0\jre\lib\rt.jar（其内容应根据实际情况变化）

下面编写一个 Java 应用程序来测试 JDK 的安装情况。

将该文件命名为 HelloWorldApp.java，其源程序如下：

```
//HelloWorldApp.java
public class HelloWorldApp{           //一个应用程序
    public static void main(String args[ ] ) {
        System.out.println("Hello World!");
    }
}
```

编译运行的情况如下：

```
d:\user\chap01>javac HelloWorldApp.java
d:\user\chap01>java HelloWorldApp
```

结果在命令行窗口屏幕上显示“HelloWorld!”，如图 2-27 所示。

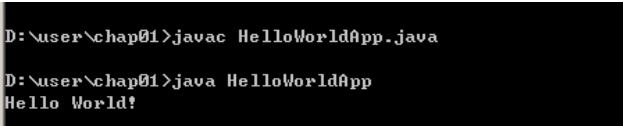


图 2-27 程序运行界面

2) 使用可执行文件安装 Tomcat

Tomcat 从 5.5 开始支持 JDK1.5，并且内置 Eclipse JDT 编译器，只需配有 JRE 即可运行，也可使用 JDK 或其他支持 Ant 的编译器。如果使用 JDK 1.4，则需安装 JDK 1.4 兼容包，或安装 Tomcat 5.0.x 版本。同时，Tomcat 5.5 支持 Servlet 2.4 和 Java Server Page（JSP）2.0 版的规范，同时又添加了许多特性。

先下载Tomcat，下载页面为 <http://tomcat.apache.org/download-55.cgi>，选择版本 5.5.27，如图 2-28 所示。

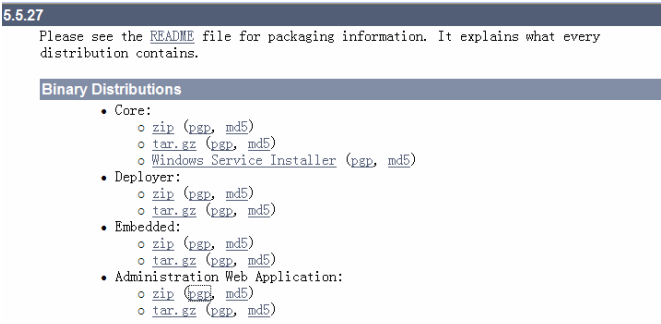


图 2-28 下载 Tomcat 5.5

其中，Core 下面的 zip 项为 apache-tomcat-5.5.27.zip（可以通过连接属性查看），是绿色软件，解压即可使用（可以使用 bin\startup.bat 启动），而 Windows Service Installer 项为 apache-tomcat-5.5.27.exe，是一个安装软件，可下载 apache-tomcat-5.5.27.exe，同时下载 Administration Web Application\zip\apache-tomcat-5.5.27-admin.zip，该包是 Web 应用程序的管理软件。

（1）安装 Tomcat。

① 运行 apache-tomcat-5.5.27.exe 并按照提示安装，如图 2-29 所示，选择 Service，作为 Windows 服务来运行。

如果要改变安装路径，可以修改安装路径中的地址，如选择安装在 D:\Tomcat 5.5 目录下，如图 2-30 所示。

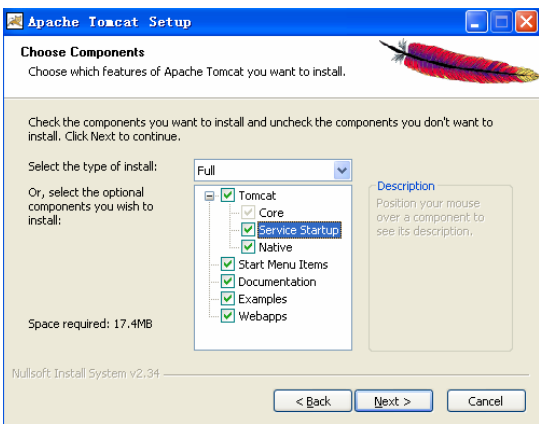


图 2-29 选择 Tomcat 的安装内容

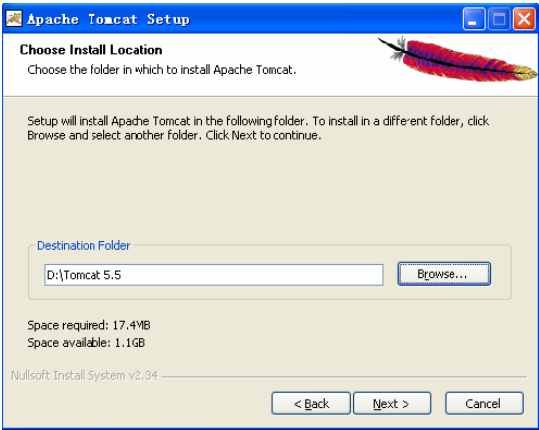


图 2-30 选择 Tomcat 的安装目录

② 设置 Tomcat 使用的端口以及 Web 管理界面的用户名和密码，确保该端口未被其他程序占用，如图 2-31 所示。

③ 选择 JVM 的安装路径，安装程序会自动搜索，如果没有正确显示，则可以手工修改，这里改为 D:\Program Files\Java\jre1.5.0_09，如图 2-32 所示。

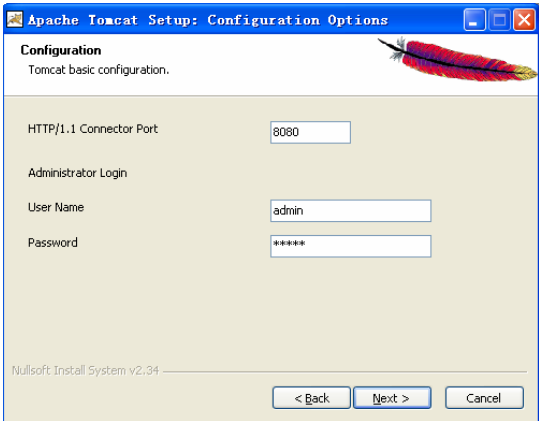


图 2-31 输入管理员密码

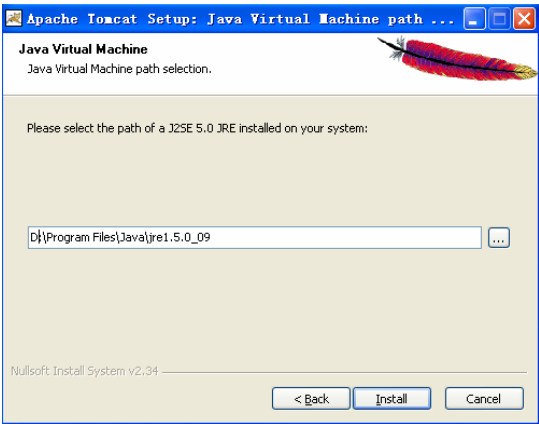


图 2-32 修改 JVM 的安装路径

④ 文件复制并成功安装后，程序会提示启动 Tomcat，如图 2-33 所示，并查看 readme 文档。Tomcat 正常启动后会在系统栏加载图标。

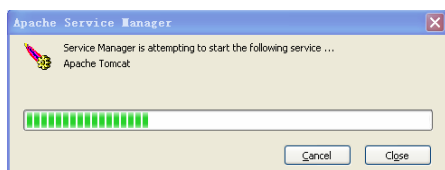
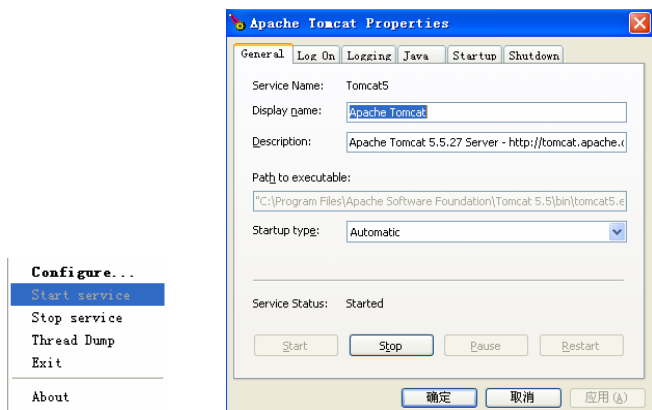


图 2-33 Tomcat 启动

在 Tomcat 的图标上单击鼠标右键可以看到一些设置项目。在如图 2-34 (a) 所示的选项框中，单击“Configure”命令可手动启动 Tomcat。

另外，也可在如图 2-34 (b) 所示的界面中设置自动启动，这样单击“Start”或“Stop”按钮即可控制 Tomcat 的运行。



(a) Tomcat 手动启动

(b) Tomcat 自动启动配置

图 2-34 启动 Tomcat 的两种方法

⑤ 设置环境变量，内容如下：

```
Java_home = d:\jdk1.5.0
Path = d:\jdk1.5.0\bin;%path%
Tomcat_home = d:\Tomcat5.5
Classpath = .;%Java_home%\lib\dt.jar;%Java_home%\lib\tools.jar;
           % Tomcat_home%\common\lib\servlet-api.jar;
           % Tomcat_home%\common\lib\jsp-api.jar;
```

至此，安装与配置都已完成，打开浏览器输入“http://localhost:8080”，即可看到Tomcat的相关信息，如图 2-35 所示。

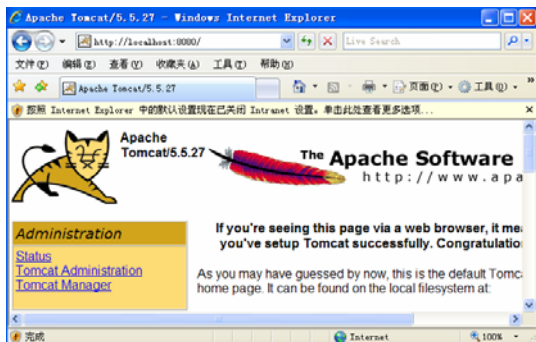


图 2-35 Tomcat 首页

⑥ Tomcat 安装后的目录结构如表 2-3 所示。

表 2-3 Tomcat 的目录结构

目 录	描 述
/bin	存放 Windows 或 Linux 平台上启动和关闭 Tomcat 的脚本文件
/conf	存放 Tomcat 服务器的各种配置文件，其中最重要的是 server.xml
/server	包含 3 个子目录：classes、lib 和 webapps
/server/lib	存放 Tomcat 服务器所需的各种 JAR 文件
/server/webapps	存放 Tomcat 自带的两个 Web 应用：admin 应用和 manager 应用
/common/lib	存放 Tomcat 服务器以及所有 Web 应用都可以访问的 JAR 文件
/shared/lib	存放所有 Web 应用都可以访问的 JAR 文件（但是不能被 Tomcat 服务器访问）
/logs	存放 Tomcat 的日志文件
/webapps	当发布 Web 应用时，默认情况下把 Web 应用文件放于此目录
/work	Tomcat 把由 JSP 生成的 Servlet 放于此目录下
假设在<CATALINA_HOME>/webapps 下有 helloapp 的 Web 应用，结构如下：	
/helloapp	Web 应用的根目录，所有的 JSP 文件和 html 文件都在此目录下
/helloapp/WEB_INF	存放 Web 发布时的描述文件 web.xml
/helloapp/WEB_INF/class	存放各种 class 文件，Servlet 文件也存放于此目录下
/helloapp/WEB_INF/lib	存放各种 Web 应用所需要的 JAR 文件，如可以存入 JDBC 驱动程序的 JAR 文件

（2）安装 Tomcat Administration。在 Tomcat 5.5.X 开始的版本中，用户必须要下载 admin 管理包，才能进入管理界面，否则会出现以下的错误提示：

Tomcat's administration web application is no longer installed by default. Download and install the "admin" package to use it.

到 Apache 官方网站上下载 admin 管理包，这里下载的是 apache-tomcat-5.5.20-admin.zip，解压后会有两部分：文件夹——Conf 和 Server；文件——license、notice 和 release-notes。

将文件和文件夹复制粘贴到 Tomcat_Home 目录下，其他不需要修改，重新启动 Tomcat 5.5.20 一次。

那么，在浏览器中输入以下的地址：

http://localhost:8080/admin

当初的错误就没有了，取而代之的是一个能输入用户名和密码的界面，输入用户在安装 Tomcat 5 时输入的用户名和密码即可，如图 2-36 所示。

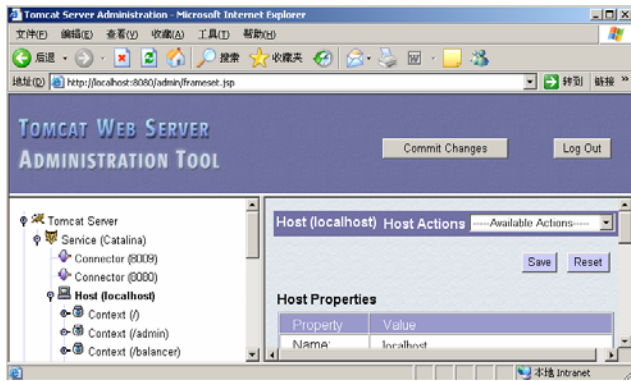


图 2-36 Tomcat 的管理员界面

(3) 添加 Context 元素。要发布 Tomcat 应用程序，添加 Context 元素，有下面几种方法，假定应用程序位于 d:\project1 目录下。

① 修改 server.xml 文件。通过在 server.xml 文件中的 Host 元素下添加 Context 元素来实现。Context 元素在 server.xml 文件中代表一个 Web 应用，一个 Host 元素中可以有多个 Context 元素。

例如，要添加 project1 应用，可以在<Host>元素和</Host>元素之间添加如下的代码：

```
<Context crossContext = "true" debug = "5" docBase = "d:/project1"
  path = "/project1" reloadable = "true"
  workDir = "work\Catalina\localhost\project1">
```

<Context>元素中各个属性表示的含义可以参考软件自带的帮助文件，这里不做详细介绍。

② 使用 Context 片断。使用 Context 片断不需要修改 server.xml 文件。使用 Context 片断和修改 server.xml 文件一样，也使用创建一个 Context 元素的方式来实现添加一个 Web 应用的目的。Context 元素的创建方式也与之相同，只是存放在位置不一样，例如，要添加 project1 应用，可以创建一个名为 project1.xml 的文件，然后把文件保存到<TOMCAT_HOME>\conf\Catalina\localhost 目录下。这个文件的内容如下：

```
<? Xml version = "1.0" encoding = "UTF-8"?>
<Context crossContext = "true" debug = "5" docBase = "D:/project1"
  path = "/project1" reload = "true">
</Context>
```

③ 使用默认发布目录。使用默认发布目录发布 Web 应用是发布 Web 应用最简单的方式，只要把 Web 应用的目录 d:\project1 复制到<TOMCAT_HOME>\webapps 目录下即可。这种方法可以支持热部署，适合较简单的 Web 应用程序。

2.4.3 Eclipse与Tomcat集成

Eclipse SDK 和 WTP 项目集成后，就可以开发 Servlet、JSP 等 Java Web 程序，开发好的 Servlet 和 JSP 需要发布到一个 Web 服务器上进行测试。本节讲解一些 Eclipse 和 Tomcat 5.5 的集成，完成 Eclipse 中的设置。

(1) 启动 Eclipse，选择菜单命令“Window”→“Preferences”，显示 Eclipse 配置对话框，单击左边目录树中的“Server”→“Installed Runtimes”选项，如图 2-37 所示。

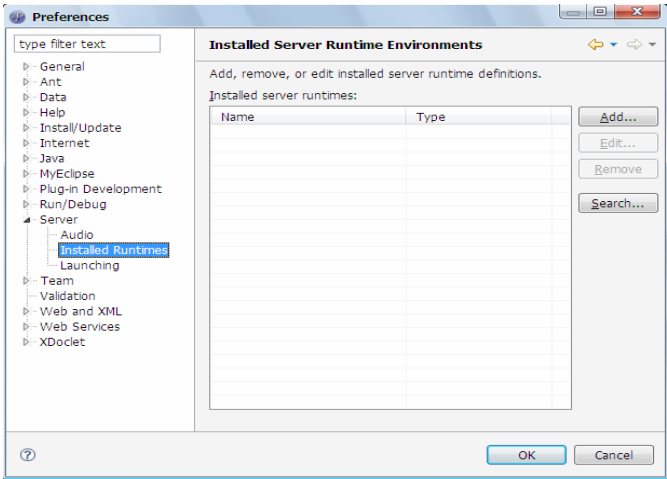


图 2-37 Eclipse 中的服务器配置

(2) 单击右边的“Add”按钮，显示“New Server Runtime”（新建服务器运行时环境）对话框，如图 2-38 所示，选择“Apache Tomcat v5.5”。

(3) 单击“Next”按钮，显示 Tomcat 服务器配置对话框，在这里设置 Tomcat 服务器的名字，选择 Tomcat 的安装路径（前提是已经安装了 Tomcat 5.5），选择使用的 JRE，如图 2-39 所示。配置完成后，单击“Finish”按钮，关闭对话框后，界面如图 2-40 所示，说明 Tomcat 服务器已经配置好了，可以用于测试 Servlet 和 JSP。

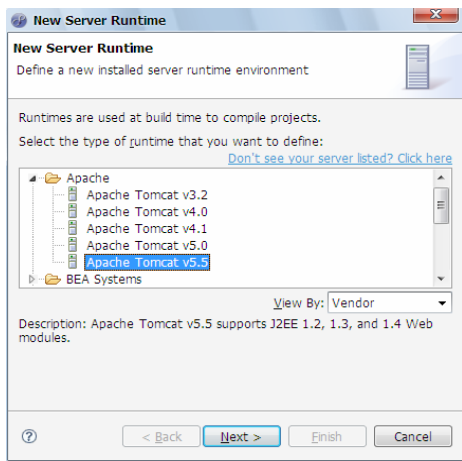


图 2-38 选择 Apache Tomcat v5.5

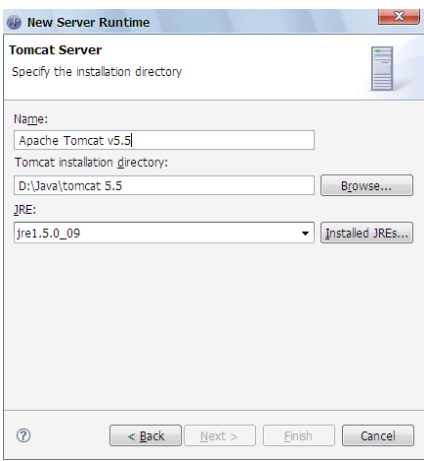


图 2-39 配置 Tomcat 服务器

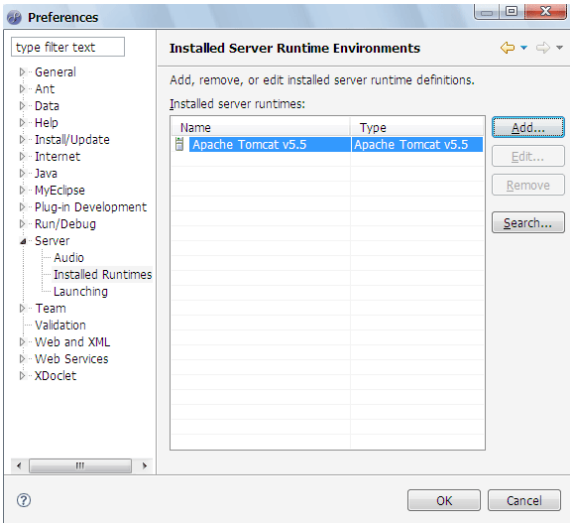


图 2-40 Eclipse 中已经配置好的服务器运行环境

如果在 Java Web 程序开发中还需要其他服务器，也可以按照这个方法进行相应配置。

2.4.4 JBoss应用服务器

1. JBoss概述

近年来，在 Java EE 应用服务器领域，JBoss 是发展最迅速的应用服务器。JBoss 是免费的，是开放源代码 Java EE 的实现。它通过 LGPL 许可证进行发布，这使得 JBoss 广为流

行。JBoss 是一个运行 EJB 的 Java EE 应用服务器，如数据库访问（JDBC）、交易（JTA/JTS）、消息机制（JMS）、命名机制（JNDI）和管理支持（JMX）。

JBoss 是开放源代码的项目，遵循最新的 Java EE 规范。例如，JBoss 发布版 4.2 实现了 EJB 3.0 的标准、JMS 1.0.1、Servlet 2.5、JSP 2.1、JMX 1.0、JNDI 1.0、JDBC 3.0 和 3.0 扩充、JavaMail/JAF、JTA1.1 和 JAAS1.0 标准。

从 JBoss 项目开始至今，它已经从一个 EJB 容器发展成为一个基于 Java EE 的一个 Web 操作系统（Operating System for Web），它体现了 Java EE 规范中最新的技术。

JBoss 应用服务器还具有以下的特点：

- （1）它将具有革命性的 JMX 微内核服务作为其总线结构。
- （2）它本身就是面向服务的架构（Service-Oriented Architecture，SOA）。
- （3）它具有统一的类装载器，从而能够实现应用的热部署和热卸载能力。

因此，它是高度模块化的和松耦合的。另外，JBoss 应用服务是健壮的、高质量的，而且还具有良好的性能。

为满足企业级市场日益增长的需求，JBoss 公司从 2003 年开始就推出了 24×7 专业级产品支持服务。同时，为拓展 JBoss 的企业级市场，JBoss 公司还签订了许多渠道合作伙伴。在后来，JBoss 公司宣布 JBoss 应用服务器通过了 Sun 公司的 Java EE 认证。这是 JBoss 应用服务器发展史上至今为止最重要的里程碑。与此同时，JBoss 一直在紧跟最新的 Java EE 规范，而且在某些技术领域引领 Java EE 规范的开发。

因此，无论在商业领域，还是在开源社区，JBoss 成为了第一个通过 Java EE 认证的主流应用服务器。现在，JBoss 应用服务器已经真正发展成具有企业强度（即支持关键级任务的应用）的应用服务器。

近年来，Hibernate 已经成为了事实上的持久化引擎。JBoss 公司致力于将自身发展成为开源项目的社区。最新版的 JBoss 应用服务器已经将 Hibernate 集成为 JMXMBean 服务，这使得用户能够在应用服务器环境中直接使用 Hibernate，而不管它是否处于 Java EE 上下文中。在最新版的 JBoss 应用服务器中，用户能够直接通过 JMXMBean 服务访问到 JBossCache 提供的服务。下一代的 JBoss 应用服务器将提供大量的新功能。除了支持最新的 EJB 3.0 规范外，新版的 JBoss AOP 同期正式发布。

同时，JBoss 开发团队还计划开发新的微内核层，即独立于 JMX，使得它能够独立使用。

JBoss 主要模块如下：

（1）JBoss EJB 容器是 JBoss 服务器的核心实现。它有两个特性，第一是在运行期产生 EJB 对象的 Stub 和 Skeleton 类；第二是支持热部署。

（2）JBossNS 是 JBoss 命名服务，用来定位对象和资源，它实现了 JNDI Java EE 规范。

（3）JBossTX 是由 JTA/JTS 支持的交易管理控制。

（4）部署服务支持 EJB（JAR）、Web 应用文档（WAR）和企业级应用文档（EARS）的部署。它会时刻关心 Java EE 应用的 URL 情况，一旦它们被改变或出现的时候将自动部署。

（5）JBossMQ 是 Java 消息规范（JMS）的实现。

（6）JBossSX 支持基于 JAAS 的或不支持 JAAS 机制的安全实现。

（7）JBossCX 实现了部分 JCA 的功能。JCA 制定了 Java EE 应用组件如何访问基于链接的资源。

（8）Web 服务器支持 Web 容器和 Servlet 引擎。JBoss 4.2 版本支持 Tomcat 5.5，Tomcat 5.0.x 和 Jetty 5.x 服务。

2. JBoss的安装

安装 JBoss 时首先要安装 JDK (仅仅安装 JRE 是不行的, 因为 JSP 页面需要编译), 本书将使用 JDK 5.0。然后把 JBoss 的压缩包解压到一个目录下, 目录名一般是“jboss_版本号”。下面的操作步骤均针对 JBoss 5.0.0 Beta1 进行配置。

1) JBoss 的下载及安装

JBoss有两类下载包, 一个是单独的, 另一个是和Tomcat集成的下载包。本章将采用前一种, 下载页面为: <http://labs.jboss.com/portal/jboss/download>, 选择 JBoss Application Server。需要的环境变量有 JAVA_HOME 和 JBOSS_HOME。环境变量 JAVA_HOME 设置为 JDK 的安装路径, JBOSS_HOME 设置为 JBoss 解压后的路径。设置方法如下:

在 Windows XP 中用鼠标右键单击“我的电脑”, 选择“属性”→“高级”选项卡, 单击“环境变量”按钮, 再单击系统变量的“新建”按钮, 然后在对话框中输入“变量名”为“JAVA_HOME”, 变量值为 JDK 的安装路径, 最后单击“确定”按钮, 用同样的方法添加 JBOSS_HOME。直接执行 JBoss\bin 目录下的 run.bat 批处理文件即可, 启动时间为 20s~1min, 视 CPU 速度和内存大小而定。

最好不要直接关闭运行 JBoss 时的控制台窗口, 直接关闭控制台可能导致 JBoss 下次启动时出现异常。正常关闭 JBoss 的方法是另外打开一个控制台窗口, 执行 JBoss\bin 目录下的 shutdown.bat 批处理文件。不过要带一个参数“-S”, 注意字母要大写。为了方便起见, 也可以创建一个快捷方式。

2) JBoss 的目录结构

JBoss 的目录结构, 如图 2-41 所示。

(1) bin 目录: 该目录包含各种脚本文件以及相关文件, 前面已经用过 run.bat 和 shutdown.bat 两个批处理文件。

(2) client 目录: 存储配置信息和可能被 Java 客户端应用程序或外部 Web 容器用到的 JAR 文件。

(3) docs 目录: 保存在 JBoss 中引用到的 XML 文件和 DTD 文件 (这里也提供了在 JBoss 中如何写配置文件的例子)。该目录下有针对性的数据库 (如 MySQL、Oracle、SQL Server 等) 配置数据源的 JCA 配置文件。

(4) lib 目录: 这里存储运行 JBoss 微内核所需的 JAR 文件。该目录下不要存储任何用户自己的 JAR 文件。

(5) server 目录: 这里的每一个子目录对应着一个服务器配置。该配置由运行脚本文件时的参数“-c<配置名称>”来确定。在 server 目录下有 3 个配置例子, 即 all、default 和 minimal, 每一个配置安装的服务都不一样, 其中 default 接受默认配置。

① minimal 目录: 仅加载启动 JBoss 所需的最少服务, 如日志服务、JNDI 和 URL 部署扫描器 (发现新的部署), 不包含 Web 容器、EJB 和 JMS。

② all 目录: 启动所有的服务, 包括 RMI/IIOP、集群服务和 Web 服务部署器 (默认配置不会被加载)。

启动 JBoss 时, 如果 run.bat 不带任何参数, 则使用的是 server/default 目录下的配置。如果要以其他目录下的配置启动 JBoss, 可以使用如下参数:

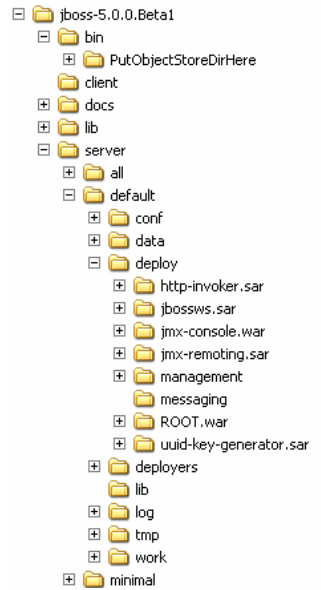


图 2-41 JBoss 安装目录

run -c all

上述命令将以 all 目录下的配置信息启动 JBoss，也可以在 server 目录下新建目录，按自己的需要写配置文件。

3) 实例讲解

下面以 default 目录为例，介绍服务器配置的目录结构。

(1) conf 目录：该目录下有指定核心服务的 jboss-service.xml 文件，也可以放置其他服务配置的文件。

(2) data 目录：该目录是 JBoss 内置的数据库 Hypersonic 存储数据的地方，也是 JbossMQ (the JBoss implementation of JMS) 存储相关信息的地方。

(3) deploy 目录：这是部署 Java EE 应用程序 (JAR、WAR 和 EAR 文件) 的位置，只需将相应文件复制到该目录下即可。该目录也用来热部署服务和 JCA 资源适配器。已经有一些服务部署到这个目录了，如 jmx-console，启动 JBoss 后即可访问。JBoss 会周期性地扫描 deploy 目录，当有任何组件改变，JBoss 会重新部署该程序。

(4) lib 目录：存放服务器配置所需的 JAR 文件。例如，用户可以将 JDBC 驱动程序放在该目录下。

(5) log 目录：存放日志信息。JBoss 使用 Jakarta log4j 包存储日志，在程序中用户也可以直接使用该信息。

(6) tmp 目录：存储在部署过程中解压时产生的临时文件。

(7) work 目录：Tomcat 编译 JSP 文件时的工作目录。

目录 data、log、tmp 和 work 在 JBoss 安装后并不存在，当 JBoss 运行时自动建立。

4) WAR 文件的部署

JBoss 支持 WAR 文件的部署。也就是说，WAR 文件部署到服务器上后不需要重新启动 JBoss (Tomcat 不支持这种特性)。WAR 文件的部署很简单，直接将 WAR 文件复制到 JBoss\server\default\deploy 目录下即可。

3. JBoss运行界面

首先确定 JBoss 已经启动，然后打开一个 IE 浏览器窗口，在地址栏中输入 “http://localhost:8080/”，就可以看到如图 2-42 所示的界面。

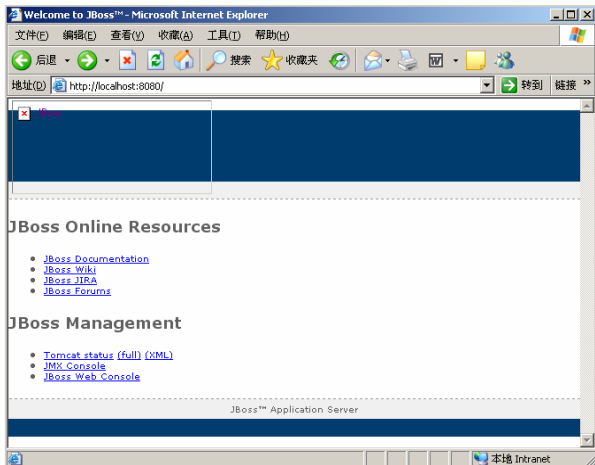


图 2-42 JBoss 运行窗口

打开一个 IE 浏览器窗口，在地址栏中输入“http://localhost:8080/jmx-console/”，这时会出现 JBoss 的 jmx-console 管理窗口，如图 2-43 所示。

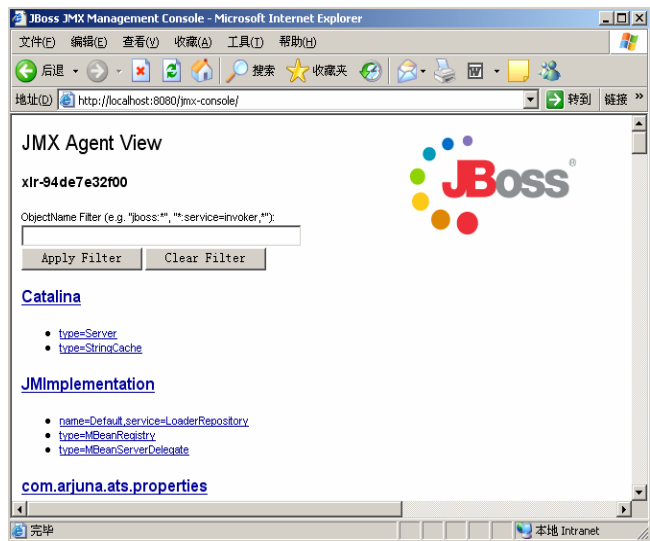


图 2-43 JBoss 的 jmx-console 管理窗口

2.4.5 Eclipse与JBoss集成——JBossIDE

JBossIDE是一系列的Eclipse插件，使用这些插件后，就可以在Eclipse中开发EJB并部署到JBoss服务器。JBossIDE可以从JBoss的网站下载，JBoss网站提供了几种选择，这里只下载插件包，地址是 http://labs.jboss.com/jbosside/download/index.html，如图 2-44 所示。

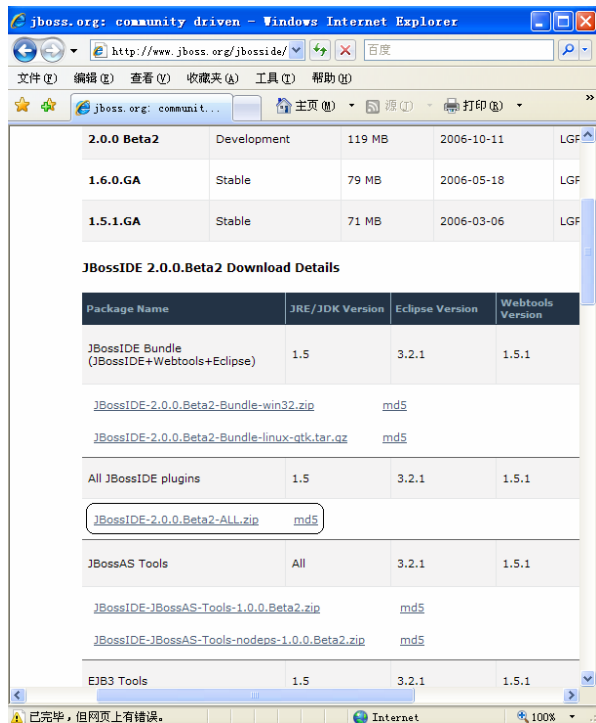


图 2-44 下载 JBossIDE

下载后，得到一个压缩文件 JBossIDE-2.0.0.Beta2All.zip，把它安装到 Eclipse 3.2 中即可。本插件的安装过程实际上就是将其解压缩到 Eclipse 的安装目录。

安装插件后，就可以在 Eclipse 中配置 JBoss 服务器的调试环境。

(1) 在 Eclipse 开发界面中，显示 JBoss Server View 视图，可以在 Eclipse 的菜单栏中单击“Window”→“Show View”→“Other”命令，在弹出的“Show View”（显示视图）对话框中选择 Server 文件夹下的“JBoss Server View”，如图 2-45 所示。

单击“OK”按钮后，JBoss Server View 视图将出现在“Eclipse”开发界面的下部，如图 2-46 所示。

(2) 建立一个 JBoss Server。在“JBoss Server View”视图上边白色区域单击鼠标右键，在出现的菜单中，单击“New”→“Server”命令，如图 2-47 所示。

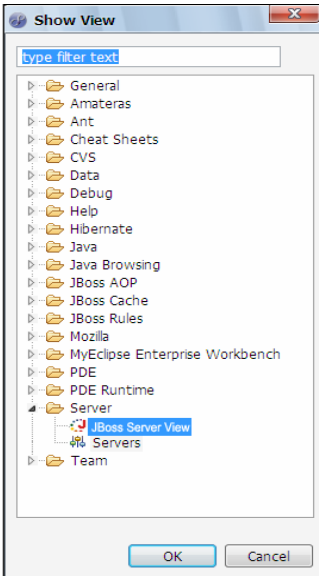


图 2-45 显示 JBoss 服务器视图

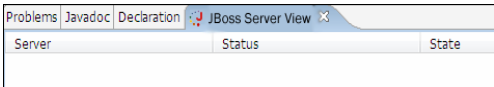


图 2-46 JBoss 服务器视图

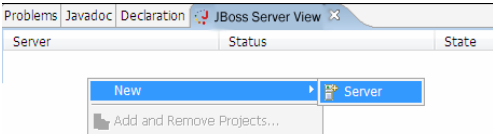


图 2-47 在 JBoss 服务器视图创建服务器实例

(3) 单击“Server”命令后弹出如图 2-48 所示的对话框，在这个对话框中，选择“JBoss v 4.0”，其他参数采用默认值。

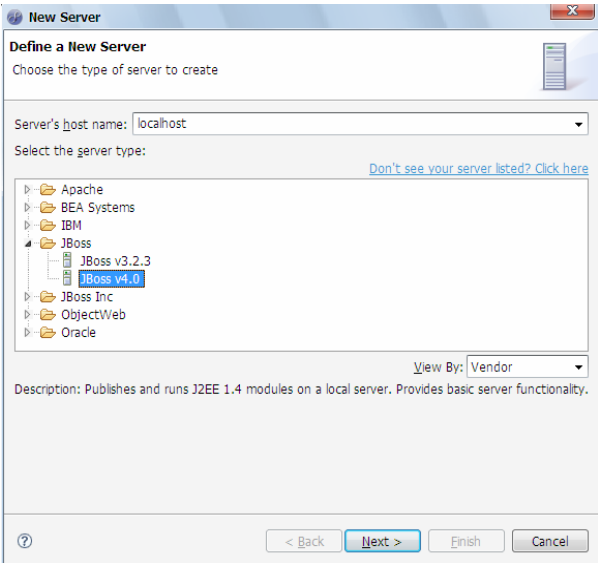


图 2-48 创建 JBoss v4.0 服务器对象

(4) 单击“Next”按钮后，弹出如图 2-49 所示的对话框，在其中选择 JBoss 服务器所在的目录。

(5) 单击“Next”按钮后，弹出如图 2-50 所示的对话框，在“Server Configuration”中选择“all”，其他采用默认值。

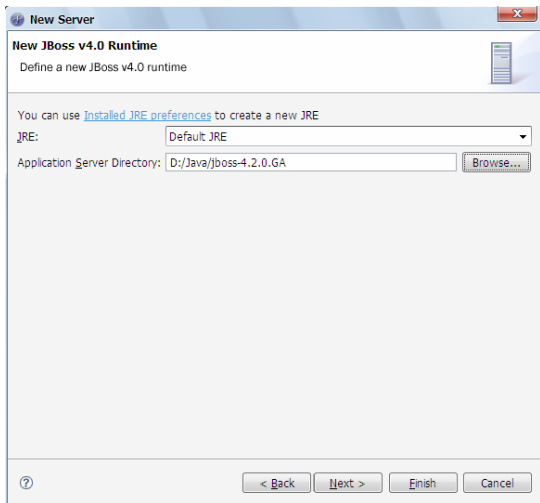


图 2-49 选择 JBoss 的安装目录

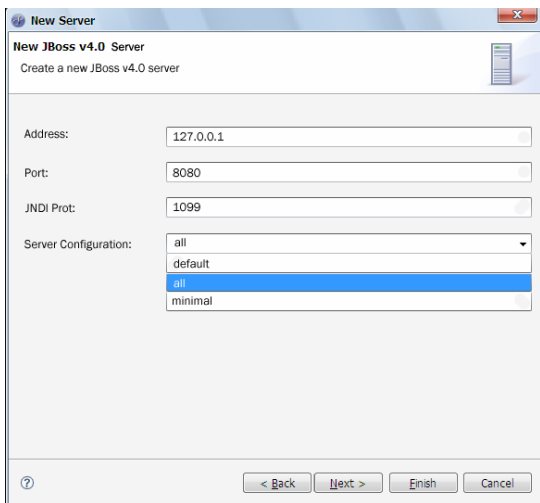


图 2-50 选择 all 服务器配置

(6) 单击“Finish”按钮，“JBoss Server View”视图如图 2-51 所示。

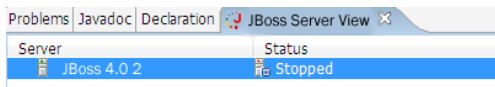


图 2-51 创建后的 JBoss 服务器对象

此时的服务器处于“Stopped”状态，需要调试 EJB 3.0 时，可以以 Debug 方式启动服务器，其方法是在服务器上单击鼠标右键，在弹出的菜单中，单击“Debug”按钮，或者单击视图右边小昆虫图标。在本视图中启动 JBoss 后的，如果没有抛出 Java 异常，此时应呈现的界面如图 2-52 所示，即说明启动成功。

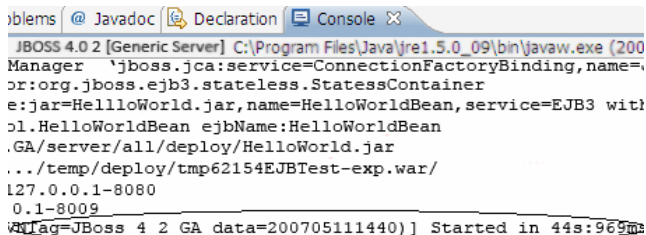


图 2-52 在 Eclipse 中启动 JBoss 服务器

通常情况下，以正常方式启动 JBoss 服务器。

习 题 2

1. 开源软件 Eclipse 由哪几部分组成？
2. 在 www.eclipse.org 网站上找出与开发 Java 程序有关的 Eclipse 项目。
3. 团队开发时，在 Eclipse 中，可以使用哪几种源代码管理系统？
4. 简述安装 Eclipse 插件的方法。

5. 使用 Eclipse 开发一个简单的插件。
6. 在 <http://eclipse-plugins.2y.net/eclipse/plugins.jsp>上浏览常用的Eclipse插件。
7. 什么是 Web 服务器？什么是应用服务器？
8. 在 Tomcat 中如何部署 Web App？
9. 简述 Eclipse 与 Tomcat 的集成。
10. 简述 JBoss 服务器的特点。
11. 简述 Eclipse 与 JBoss 的集成。

第 3 章 Java Applet及JDBC

本章介绍了 Java EE 规范中的 Java Applet（小程序）及 JDBC，为后面章节学习提供基础。

3.1 Java Applet基础

3.1.1 在HTML中调用Applet

编写一个 Java 应用程序时，应该注意以下几点：

- （1）用 Java 语言编写一个应用程序，然后以.java 为扩展名将其保存。
- （2）用 Java 命令将此应用程序编译为字节码，然后以.class 为文件扩展名保存。
- （3）用 Java 命令解释并执行.class 文件。

应用程序是独立的程序，对比之下，Applet 是嵌入到其他程序中的一段 Java 小程序，可以在网页中运行 Applet，或者在来自于 Java 开发者所附带的配套工具 AppletViewer 中运行。Applet 必须被嵌入到 HTML 当中，也就是嵌入到超文本标记语言中。HTML 是一种简单的被用来为因特网生成网页的语言，当创建 Applet 时，应该做到以下几点：

- （1）用 Java 语言编写 Applet，并以.java 为扩展名将其保存，就像编写应用程序一样。
- （2）用 Java 命令将 Applet 编译为字节码，就像编译应用程序一样。
- （3）编写一个 HTML 文件，它含有一条语句调用 Applet 编译的.class 后缀文件。

（4）将此 HTML 文件加载到网页浏览器中（如 Navigator 或 Internet Explorer），或者运行 AppletViewer 程序，它反过来使用 HTML 文件。

通常 Java，特别是 Applet，在程序员中是很受欢迎的，主要是因为用户可以在因特网上用网页浏览器来运行 Applet，网页浏览器是用来让 HTML 文件显示在显示器端的程序。网页文件通常包括一些 Java Applet。

HTML 包含一些在网页中能使文件布局合理的命令，输出生动的图像，并且能使网页连接到其他网页。运行一个 Java Applet，不需要掌握全部的 HTML 语言，只需要学习叫做 tag（标签）的两个 HTML 命令对。

HTML 文件都是以标签<HTML>开头的，像所有的标签一样，这种标签用单括号括起来，关键字“HTML”用来指明以下的部分是 HTML 文件。每一个 HTML 文件都是以<HTML>为结尾的，在任何标签前插一个符号“/”表明这是该标签对的另一半标签的结束。

下面是一个最简单的 HTML 文件：

```
<HTML>
</HTML>
```

注意：与 Java 编程语言不同的是，HTML 语言不区分大小写，所以将<HTML>写为<html>是允许的。与 Java 语言一样，HTML 语言忽略了空白，所以可以将 HTML 文件写成一行，如<HTML></HTML>。

这种简单的文件开头和结尾对程序的运行结果不产生影响，在 Java 中输入一个开始

花括号紧接着一个结束花括号的方式来建成的一个简单程序的情形也是如此。当然，HTML 包含更多的语句。例如，为了在 HTML 文件中运行 Applet，标签<Applet></Applet >需成对出现。通常<Applet>中含有三个变量：CODE，WIDTH，HEIGHT。示例：

```
<APPLET CODE = "aClass.class"WIDTH = 300 HEIGHT = 200>
```

CODE：要调用的编译了的 Applet 的名字。

WIDTH：Applet 在显示器屏幕上的宽度。

HEIGHT：Applet 在显示器屏幕上的高度。

所调用的 Applet 名字必须是已编译的 Java Applet（以.class 为扩展名）。Applet 的宽和高是以 pixel（像素）衡量的，pixel 就是图像单元，或者说是组成显示器的细小的光点。很多显示器的规格是 640 像素×480 像素，所以语句 WIDTH=300、HEIGHT=200 可生成占据显示器 1/4 大小的 Applet（显示器高和宽的一半）。

注意：一个 VGA 显示器的规格是 640 像素×480 像素。一个 SVGA 显示器的规格可以达到 1280 像素×1024 像素，一般的 Applet 的最大尺寸应该设置为 600 像素×400 像素，这样可以确保大部分人能看到整个的 Applet。

接下来可以建立一个简单的 HTML 文件，用来显示在下面建立的 Applet，将此 Applet 命名为 Greet，它将占据 450 像素×200 像素。

为创建一个简单的 HTML 文件，应执行以下操作。

（1）打开文本编辑器。

（2）输入 HTML 标签<HTML>。

（3）在下一行中输入 Applet 的开始标签、名字及尺寸规格：

```
<Applet CODE:"Greet.class"WIDTH = 450 HEIGHT = 200>
```

（4）在下一行中输入 Applet 的结束标签：</Applet>。

（5）接着输入 HTML 的结束标签：</HTML>。

（6）将该文件以 Test.html 为名保存在文件夹中。就如同编写 Java 应用程序一样，确信将此文件保存为文本格式。后缀名不要求，若加上更好，如果使用记事本或其他文本编辑器，一定要以.html 为扩展名，如“A:\Chapter.03\Test.html”。

3.1.2 编写一个Applet

编写一个 Applet 仅涉及到学习比写一个 Java 应用程序稍多的一点变化的内容，要编写一个 Applet 要做到以下几点：

（1）添加某些 Import（导入）语句。

（2）学会应用 Windows 组件和 Applet 方法。

（3）学习使用关键字 extends。

学习 Java 程序设计时，已经会使用一条 Import 语句来访问类。例如，在应用程序中的 java.util.date，导入日期类的目的在于避免重复编写日期处理子程序。同样，Java 的开发者定义了许多类用来处理一般的 Applet 的需求。大多数 Applet 至少包中含了两条导入语句：“import java.applet.*;”和“import java.awt.*;”。这种 java.applet 包中包含称做 Applet 的一个类。java.awt 包的全称是 Abstract Windows Toolkit 或者 AWT。它通常含有可使用的 Windows 组件，如标签、菜单和按钮等，当导入 java.awt 时，不需要重新编写所需的组件。Java Applet 可不要求包含 Windows 组件，但通常确实会用到很多。

例如，一个最简单的 Windows 组件 Label 是内置的标签类，可以在 Applet 中显示文本信息。Label 类也包含用来指示其特征信息的属性，如字体和排列。如同使用其他对象一样，可以声明一个 Label 而不用分配存储空间。例如，在“Label greeting;”中，或者不用任何自变量调用标签结构，例如，“Label greeting = new Label();”，再使用 setText()方法能将一些文本赋于标签中，例如，“greeting.setText("Hi there");”。此时，可以传递给该标签一个字符串自变量，故这个标签在结构上被初始化，例如，“Label.greeting = new Label("Hello, who are you");”。向一个 Applet 窗口中添加一个组件的方法是 add()，例如，如果一个标签被定义为“Label.greeting=new label("Hello, who are you? ");”，则可以用 add(greeting)命令将 greeting 添加到 Applet 中。

注意：add()方法的对象是 Applet 自身，所以当向窗体中添加组件时，可以写做“this.add();”代替“add();”。

当建立一个应用程序时，应先写使用 class 头的语句，如“Public class aclass”。Applet 像 Java 应用程序一样也是以同样的方式开头的，但是必须包含 extends Applet 子句，关键字 extends 指示所编写的 Applet 将建立在或者继承于 Java Applet 包中所定义的 Applet 类的特性上。

Applet 类提供了在 Applet 运行时任何 Web 浏览器所使用的一般情况。在一个应用程序中，main()方法调用了所写的其他方法，对于一个 Applet，浏览器可以自动调用许多方法。下面是每个 Applet 中须包含的四种方法：

```
public void init()
public void start()
public void stop()
public void destroy()
```

如果在程序中未编写一个或多个上述方法，Java 会自动建立这些方法。Java 生成的只有一对尖括号，也就是说，它们是空的。为了建立一个有实际意义的 Java 程序，必须至少在以上方法的一种中编写自己的代码。

Init()方法在 Applet 中被首先调用，可以用来初始化任务，如设置变量初值，或将 Applet 组件放置在屏幕上。必须将代码 init()头，写做：public void init()。

下面给出在屏幕上显示“Hello,who are you?”的 Applet 源程序。

```
import java.applet.*;
import java.awt.*;
public class Greet extends Applets
{
    Label greeting = new Label ("Hello,Who are you? ");
    Public void init()
    {
        add (greeting);
    }
}
```

接下来，将建立并编辑此 Greet Applet。

为建立并运行 Greet Applet，应该执行以下操作。

- (1) 在文本编辑器中打开一个新文本。
- (2) 输入此例所示的代码。

(3) 将此文件以 Greet.java 为名保存在 “C:” 盘上。

(4) 用 Java Greet.java 命令编译该程序。

(5) 如果需要, 纠正错误, 再次进行编译。

为了能运行 Greet Applet, 可以用 Web 浏览器或 AppletViewer 命令。

在下面的步骤中, 将涉及以上两种方式:

为使用 Web 浏览器运行 Applet, 应执行以下操作:

(1) 打开 Web 浏览器, 如 IE 或 Navigator。不需要连接到因特网, 可以使用本地的浏览器。

注意: 可以双击桌面上的快捷方式图标, 或使用开始按钮启动 Web 浏览器。如果在启动 Web 浏览器时遇到困难, 可以寻求技术人员的帮助。

(2) 单击菜单栏的 “File”, 单击其 “Open” 或者 “Open page”。然后在地址栏中键入 “C:\Chapter03\Text.html”, 也就是前面所建立的 Greeting.class 的 HTML 文件的全路径。按 Enter 键, 显示屏中出现 Applet 的运行结果: “Hello, who are you?”。

(3) 通过单击浏览器程序窗口右上角的 “关闭” 按钮关闭浏览器, 也可以使用 Applet Viewer 命令测试所编写的 Applet。

注意: 有些 Applet 不能在浏览器中正常运行。Java 被设置具有一些安全特征, 所以当 Applet 在因特网上运行时, Applet 不能执行恶意的任务, 如从磁盘上删除数据等。如果 Applet 没有危害安全性, 则使用 Web 浏览器或 Applet Viewer 命令测试它将会得到相同结果。

为使用 AppletViewer 命令运行 Applet, 应执行以下操作。

(1) 在命令行中键入 “AppletViewer test.html”, 按 Enter 键, AppletViewer 窗口打开并显示此 Applet。

(2) 使用鼠标拖动边框可以调整窗口的大小。注意, 当想调宽窗口时, 可以将边框向右拖动, 向左拖动时使窗口变窄。

(3) 单击 “关闭” 按钮将 AppletViewer 窗口关闭。

3.1.3 改变标签的字体

如果使用 Web 浏览器访问 Web 页, 可能对 Greet.java Applet 印象不深, 会觉得字符串 “Hello, who are you?” 的字体很平淡, 无生气。幸运的是, Java 提供了 Font 对象, 它包括 typeface (字体) 和 size (大小) 信息。setFont() 方法需要 Font 对象自变量。为了构建一个 Font 对象, 需要设置三个自变量: Typeface, Style, Point size。

Typeface 是代表字体的字符串, 通常包括宋体、楷体、黑体和隶书, 以及 Arial、Helvetica、Courier、TimesRoman 等字体。Typeface 只是一个要求, 运行 Applet 的系统也许并不支持所要求的某字体。因此往往用默认字体。Style 是显示文本风格的属性。它主要有 Font.BOLD (粗体), Font.ITALIC (斜体), Font.PLAIN (普通) 三种风格。Point size 是一个代表 1/72 英尺的整型数。通常的印刷体文本为 12points, 头标题为 30 points。

为了给 Label 对象定义新的字体, 可以建立 Font 对象, 例如: “Font headlineFont = new Font (“Helvetica”, Font.BOLD 36);”, 然后使用语句 “greeting.setFont (headlineFont);”, 用 setFont 方法为 Label (标签) 分配字体。

要点: Typeface 的名字是字符串, 所以当声明这个 Font 对象时要用引号括起来。

接下来讲述如何改变 Greet Applet 中的字体。

(1) 打开 Greet.java 文件。

- (2) 将光标移到声明 **Greeting** 标签的那行的结尾，按 **Enter** 键进入下一行。
- (3) 通过输入如下一行语句来声明名为 **bigFont** 的一个 **Font** 对象：

```
Font bigFont = new Font("TimesRoman", Font ITALIC, 24);
```
- (4) 将光标移到 **init()** 方法的花括号的右边，然后按 **Enter** 键进入下一行。
- (5) 通过输入 “**greeting.setFont (bigFont) ;**”，将 **greeting** 的字体设置为 **bigFont**。
- (6) 以相同的文件名 **Greet.java** 将其保存。
- (7) 在命令行中使用 **Java Greet.java** 命令将该程序编译。
- (8) 通过执行 **AppletViewer test.html** 命令，运行以前建立的运用 **Applet** 的 **HTML** 文档，便可看见结果。
- (9) 关闭 **AppletViewer** 窗口。

3.1.4 向Applet 添加文本框和按钮组件

Applet 中除了包括 **Label**，通常还包括具有 **Windows** 特征的其他组件，如文本框和按钮。文本框是用户能向其中输入一整行字符的 **Windows** 组件（文本数据通常包括键盘上的任何字符及数字，标点符号）。典型的使用是，用户向文本框中输入一整行字符后按 **Enter** 键或者单击“提交”按钮进入数据。可以用以下任一种方法构建一个文本框对象。

- **Public textfiled()**，它以未定长度的方式生成空的文本框。
- **Public textfiled(int numconlumns)**，其中 **numconlumns** 指定了框的宽度。
- **Public textfiled(string initialtext)**，其中 **initialtext** 提供了框的初始文本。
- **Public textfiled(string initialtext, int numcomns)**，它指定了初始文本和宽度。

例如，为了使用户能回答“Who are you?”这个问题，可以定义一个文本框，编写“**Textfiled.answer = new Textfiled(10);**”语句建立一个能显示 10 个字符的空文本框，将此文本框命名为 **answer** 并将其添加到 **Applet** 中，可以写成 **add (answer)**。

要点：一个文本框实际显示的字符个数取决于设置的字体和输入的字符，例如，在大多数字体中，“w”比“i”要宽，所以一个文本框能显示 24 个“i”字符，却仅能显示 8 个“w”字符。

要点：建立文本框时，应实际预计用户输入的字符个数，尽管用户输入的字符个数多于能显示出的最大个数，仍能用滚动条的方式看到不在视野内的字符文本。

文本框可使用几个现存的其他方法：**setText()**方法，允许更改已经建立的文本框信息，例如：“**answer.setText("Thank you");**”；**getText()**方法允许从文本框获取字符串，例如：“**String whatDidtheysay = answer.getText()**”。

当用户遇到置于 **Applet** 中的文本框时，用户应将光标指向文本框中，以便输入文本。也就是从光标所指处输入所有来自键盘上的信息。如果能让光标自动出现在文本框而不需要用户事先单击，则可以用 **requestFocus()**方法。例如，如果已经向 **Applet** 中添加了名为 **answer** 的文本框，则 **answer.requestFocus()**会引起光标出现在文本框中，用户不需要移动鼠标而立即开始输入文本。除了可以节约用户的时间外，当建立几个文本框时，若只想让其中一个引起用户注意，也可以采用 **requestFocus()**方法。任何时候 **Windows** 中只有一个组件具有键盘焦点。

当文本框具有接收键盘输入的能力时，这个文本框是可编辑的，如果想禁止用户向其中输入信息，则可以使用 **setEditable()**来改变文本框的编辑状态。例如，如果只想使用户拥

有一次正确回答问题的机会时，可以通过运用语句“`answer.setEditable(false);`”来防止用户改变或编辑文本框中的字符信息。如果条件改变，并且想使用户拥有修改的权限，可以使用语句“`answer.setEditable(true);`”。

按钮比文本框更容易创建，这里仅有两种按钮需要构建：

- `Public Button()`，它创建一个无标签的按钮。
- `Public Button(string label)`，它创建一个带标签的按钮。

例如，为创建一个带有“`press when ready`”标签的按钮，可写语句：“`Button readyButton = new Button("press when ready");`”。为了将此按钮添加到 Applet 当中，必须写“`add (ready Button);`”语句。可以使用 `setLabel()`方法改变按钮标签，例如，“`readyButton.setLabel("Don't press me again!");`”；或者使用 `getlabel()`方法获得标签并将之赋予一个字符型对象，例如，“`string whatonButton = readyButton.getlabel();`”。

要点：确信按钮上的标签描述该键的功能。

要点：像使用文本框组件一样，可以在按钮组件上使用 `requestFocus()`方法，具有键盘焦点的按钮表面出现一个轮廓，故它突出于其他按钮。

接下来，将文本框和按钮添加到 Applet 中。

将文本框添加到 `Greet.java.applet`，应该执行以下操作。

- (1) 在文本编辑器，打开 `Greet.java` 文件。
- (2) 将光标置于定义字体对象行的结尾，按 `Enter` 键进入下一行。
- (3) 为声明一个带有“`press me`”标签的按钮及一个空文本框，输入下面内容：

```
Button pressme = new Button ("press me");
Textfiled answer = new Textfiled("", 10);
```

(4) 将光标移到把 `Greeting` 添加到 Applet 中的 `add()`语句的结尾，然后按 `Enter` 键进入下一行。

- (5) 输入下列语句向 Applet 中添加文本框和按钮：

```
Add(answer);
Add(press me);
```

- (6) 在下一行，输入下面语句以便设置此 `answer` 要求的焦点：

```
Answer.requestFocus();
```

- (7) 将该文件保存，并用 `Javac Greet.java` 编译。

(8) 用 `AppletViewer test.html` 命令运行 Applet。确定能将字符输入文本框中，并能单击该按钮，尽管此按钮现在还没有相应的事件执行程序。

- (9) 关闭 `AppletViewer` 窗口。

3.1.5 Applet的事件驱动编程

当用户使用 Applet 在组件上采取动作时，一个事件便发生了，如鼠标的单击事件。在文本中所编写的程序都是结构化的，也就是说，程序“指挥”着事件发生的顺序：接收用户的输入，写出决定和循环，然后产生输出。当接收用户输入时，不能控制用户用于完成一个响应提示所需的时间，但是能控制程序不向前进行，直到输入完成。反之，对于事件驱动的程序设计，用户可能会以任何顺序开始任何事件。例如，如果使用一个字处理程序，任何时刻在处理上会有很多种选择：可以输入字符，用鼠标选定文本，单击鼠标将文本转换为粗体，或者将字体转换为斜文，选择菜单，等等。对于建立的每个字处理文档，可以在任何时刻选择

当时看来适当的选项。字处理程序必须准备好及时响应所发生的事件。

在事件驱动的程序中，组件上发生的事件被认为是事件的源，用户单击按钮就是一个例子，而用户向文本框中输入信息是另一个源。对事件有关系的对象称为监听器。不是所有的对象都能接收事件，也可能出现单击屏幕中的若干区域而没有反应的情况。如果想使一个对象能够接收事件，例如，想将 `Applet` 设置为一个监听器，则必须将此对象登记为源的一个监听器。当今的报纸在新闻服务中心登记例如，`Associated Press` 或 `United Press International`。新闻服务中心有一张预订表，当有重要事件发生时便对每一个登记者发送消息。类似地，一个 `Java` 组件源对象（如按钮）主要维持一张登记的监听器列表，当事件发生时（如鼠标的单击事件），便通知全部已登记的监听器（如 `Applet`）。当监听器“接收新闻”时，作为监听器对象一部分的某个事件处理方法便响应事件。

注意：源对象及监听器对象可以是一个相同的对象，例如，按钮上的标签在用户单击时可以改变。

为了在任何所建立的 `Applet` 中响应用户事件，需要做到以下几点。

1. 使Applet准备接收消息

通过导入 `java.awt.event` 包在程序中，并添加子句 `implements ActionListener` 到 `class`（即类）头中，便可以使 `Applet` 准备接收鼠标事件。`java.awt.event` 包中包括 `ActionEvent`、`ComponentEvent` 和 `TextEvent` 事件类的名称。`ActionListener` 是接口，或者说是一组特定可以用来使用事件对象的方法。`Implement ActionListener` 提供了标准事件方法说明，允许 `Applet` 用 `ActionEvents` 工作（它是用户单击按钮时出现的事件类型之一）。

要点：可以通过 `Implement` 因子来鉴别诸如 `ActionListener` 那样的接口，不需要“导入”（不用写 `import java applet.*`）或“扩展”（不用写 `extends Applet`）。

2. 告诉Applet接收发生的事件

使用 `addActionListener()` 方法可以告诉 `Applet` 接收 `ActionEvents`。如果声明了一个名为 `aButton` 的按钮，并且想在用户单击时有所行动，那么 `aButton` 就是消息源，而将 `Applet` 当做目标并向其中发送消息。我们已经知道可用 `this` 来指定当前的方法，故“`aButton.addActionListener(this);`”引起任何来自 `aButton` 的 `ActionEvent` 消息（单击按钮）送到“当前 `Applet`”。

要点：不是所有的事件都带有 `addActionListener()` 方法的 `ActionEvents`。例如，`TextEvent` 有一个 `addTextListener()` 方法。

3. 告诉Applet如何响应任何发生的事件

`ActionListener` 接口包括 `actionPerformed(ActionEvent e)` 方法规范，当 `Applet` 登记作为监视器并且用户单击相应按钮时，`actionPerformed()` 方法将被执行。必须编写 `actionPerformed()` 方法，它像所有的方法那样也包含程序头及程序主体。将函数头写成 `public void actionPerformed (ActionEvent e)`，这里 `e` 是为事件（单击按钮）选择的任何名字，它启动 `ActionListener`（此 `Applet`）的通知。方法主体中包含任何当行动出现时想要执行的语句。你可能想要进行数据计算，构建新的对象，产生输出，或者执行任何其他的操作。例如，下面显示了 `actionPerformed()` 方法，它产生了一行在系统提示符后面的输出。

```
public void actionPerformed(ActionEvent someEvent)
{
    System.out.println
        ("I'm inside the actionPerformed() method!");
}
```

接下来，将通过在 Applet 中的按钮与文本框添加功能而使前述建好的 Greet.java Applet 成为事件驱动程序。

(1) 向 Applet 中添加功能。

① 在文本编辑中打开 Greet.java 文件。

② 在程序中添加第三个导入语句 “import java.awt.event.*;”。

③ 将光标移到类头 (public class Greet extends Applet) 的末尾的空格键，输入 implements ActionListener。

④ 将光标移到添加了 Press Me 按钮到 Applet 的 init() 方法的末尾，按 Enter 键。通过输入语句 “press me addActionListener(this);” 使 Applet 准备接收按钮源事件。

⑤ 将光标移至 init() 方法结束花括号的右边，按 Enter 键。在 init() 方法之后，和 Greet 类结束括号之前输入下面的 actionPerformed() 方法。这个方法声明了将保存用户名的字符串，在 answer 文本框中使用 getText() 方法获得该字符串，然后显示屏幕信息给用户。

```
public void actionPerformed(ActionEvent thisEvent)
{
    String name = answer.getText();
    System.out.println("Hi there + name")
}
```

⑥ 保存并编译该程序。在命令提示符使用 AppletViewer text.html 命令执行 test.html 程序。

⑦ 将你的名字输入到文本框，然后单击 Press Me 按钮，查看命令提示符屏幕。个人信息 (“Hi there ”和你的名字) 出现在命令提示符屏幕上。

帮助：可能需要调整 Applet Viewer 的窗口位置，以便可以看到命令行中的输出。

⑧ 使用鼠标选定文本框中的名字 (使之高亮度)，然后输入新的名字，单击 Press Me 按钮，一个新的问候将出现在命令行中。

⑨ 关闭 AppletViewer 窗口。

在大多数包括文本框的 Applet 中，有两种获得用户输入的方法。通常，可以输入文本信息，然后单击 “提交” 按钮，或者输入文本后按 Enter 键。如果 Applet 需要接收文本框中的事件信息，则需要将 Applet 登记为同文本框的事件监听器。

(2) 为了添加按 Enter 键便能接收用户在文本框中的输入的能力，应该执行以下操作。

① 在 Greet.java 文本文件中，将光标置于 “pressme.addActionListener(this);” 后边，然后按 Enter 键。

② 输入下面的语句使 answer 域接收输入：

```
answer.addActionListener(this);
```

③ 保存并编译该程序。在命令提示符后面用 AppletViewer test.html 命令运行 test.html 程序，确认通过单击 Press Me 按钮或按 Enter 键能向文本框中输入姓名。

④ 关闭 AppletViewer 窗口。

3.1.6 添加输出到一个Applet

在命令行屏幕产生输出的 Applet 并不精彩。随着各种事件的出现，会想在 Applet 做一些改变。例如，代替使用 `System.out.println()` 发送一个 `greeting` 到命令行中，添加一个 `greeting` 到 Applet 中。完成此任务的一种方式创建一个新标签，在用户输入名字后，用 `add()` 方法将该标签添加到 Applet。可以通过语句 “`label personalGreeting = new Label("");`” 声明一个新的空白标签。在姓名被接收后，可以使用 `setText()` 方法来为 `personalGreeting` 设置标签文本到 “`"Hi there"+ name`”。

为添加 `PersonalGreeting` 标签到 Applet，应该执行以下操作。

① 在 `Greet.java` 文件中，从 `actionPerformed()` 方法中去掉 `System.out.println()` 语句。

② 对 `actionPerformed()` 方法添加下列语句来声明一个名为 `personalGreeting` 的新标签，设置 `personalGreeting` 中的文本信息，然后将 `personalGreeting` 添加到 Applet 中：

```
Label personalGreeting = new label("");
personalGreeting.setText("Hi"+ name);
add(personalGreeting);
```

③ 保存并编译该程序，然后使用 `AppletViewer` 命令运行此 Applet。尝试在文本框输入一个姓名并按 `Enter` 键或者单击 `Press Me` 按钮。会发现没有反应，当向文本框中输入姓名并单击 `Press Me` 按钮，也没有反应。

④ 现在使用鼠标调整窗口边缘重置 `AppletViewer` 窗口大小，或最小化然后恢复它，于是 `personalGreeting` 标签就会出现。

⑤ 关闭 `AppletViewer` 窗口。

Applet 先不显示 `personalGreeting` 的原因是因为向其中添加的时间太晚。当 Applet 启动时，`init()` 方法激活全部的 Applet 组件，而语句 “`add(personalGreeting);`” 不是 `init()` 方法的一部分。当最小化或重置 Applet 尺寸大小时，它“知道”必须重“画”以适应新的尺寸。类似地，如果打开另一个应用程序时，原来的 Applet 界面会全部或部分隐藏，当关闭当前最顶端的窗口时，下面的 Applet 已经“知道”它必须重“画”界面尺寸。然而，当使用 `add()` 方法将新组件添加到屏幕中时，该 Applet 还没“意识”到它已经“过时”。

通过使用 `invalidate()` 方法可以使 Applet “知道”它已“过时”，因为 Applet 标注了窗口，故它“知道”，但还未用最近的变化来更新。然后，可以通过使用 `validate()` 方法引起实际变化，Applet 可重“画”任何被隐藏窗口的尺寸。

为在添加 `personalgreeting` 后重“画”`AppletViewer` 窗口的尺寸，应该执行以下操作。

① 在 `Greet.java` 文件的 `actionPerformed()` 方法中，将光标置于语句 “`add(personalGreeting);`” 后面，按 `Enter` 键并输入如下语句：

```
invalidate();
validate();
```

② 保存，编译并运行该 Applet。在输入姓名并按 `Enter` 键或 `Press Me` 按钮后，`greeting` 现在立即会显示出来。

③ 关闭 `AppletViewer` 窗口。

如果能向 Applet 中添加组件，也应该能去掉它们，使用 `remove()` 方法即可。例如，在用户输入姓名到文本框后，你可能会不想要用户再次使用该文本框或它的按钮，可以从 Applet 去掉它们。通过置组件的名称于括号中来使用 `remove()` 方法。与使用 `add()` 一样，使

用 `remove()`后必须重“画”以显示结果。

为在 `Greet Applet` 中去掉文本框及按钮，应该执行以下操作。

① 将光标置于 `Greet.java` 文件的 `acionPerformed()`方法中语句“`add(personalGreeting);`”的后面，按 `Enter` 键，并输入如下语句：

```
remove(answer);  
remove(pressMe);
```

② 保存、编译并运行该 `Applet`。当输入姓名并按 `Enter` 键或单击 `Press Me` 按钮时，文本框和按钮就会从屏幕消失。

③ 关闭 `AppletViewer` 窗口，然后关闭文本编辑器。

3.2 Applet的生命周期和更复杂的Applet

3.2.1 Applet的生命周期

`Applet` 很受欢迎是因为它能很容易地在 `Web` 页中使用，由于 `Applet` 是在浏览器中运行的，所以 `Applet` 类包含能被浏览器自动访问的方法。前面已经了解了四种方法：`int()`、`start()`、`stop()`和 `destroy()`。

在前面已经编写了自己的 `init()`方法。当编写的方法程序头与自动提供的方法一样时，必然代替或覆盖自动提供的方法。当每次含有 `Applet` 的 `Web` 页被加载到浏览器中或者当使用 `AppletViewer` 命令调用 `Applet` 的 `HIML` 文档时，除非为该 `Applet` 编写了一个 `init()`方法，否则会执行自动 `init()`方法。当有一些初始化任务需要完成时，应该编写自己的 `init()`方法，如创建用户接口组件。

`start()`方法在 `init()`方法后执行，在 `Applet` 每次成为非活动性状态之后至要恢复成活动性之前再次运行它。例如，当使用 `AppletViewer` 命令运行 `Applet` 时，如果将 `AppletViewer` 窗口最小化，那么 `Applet` 此刻为非活动性；当恢复窗口时，该 `Applet` 又会成为活动性。在因特网上，用户可以离开一个网页而去访问另一网页，然后又退回到第一个网页，就这样 `Applet` 在活动性与非活动性之间不断变换。当用户重新访问 `Applet` 时存在任何你想采取的行动，则可编写自己的 `start()`方法。例如，想恢复用户离开 `Applet` 时所挂起的程序的活动性，便可编写有相应功能的自己的 `start()`方法。

当用户离开 `Web` 页时，可能会最小化窗口或访问另一个 `Web` 页，这时 `stop()`方法被调用。仅当想对一个不再可见的 `Applet` 采取某行动时，可编写自己的 `stop()`方法去覆盖正存在的 `stop()`方法，其他情况下，通常不必编写自己的 `stop()`方法。

当用户关闭浏览器或 `AppletViewer` 时，调用 `destroy()`方法。关闭浏览器或 `Applet Viewer` 就会释放 `Applet` 占据的资源。与 `stop()`方法一样，通常不必编写自己的 `destroy()`方法。

总而言之，每个 `Applet` 都有相同的生命周期轮廓，如图 3-1 所示。当 `Applet` 运行时，`init()`方法开始执行，接着 `start()`方法开始执行。如果用户离开 `Applet` 页，则 `stop()`方法开始执行，当用户返回时，`start()`方法又开始执行。`stop()-start()`序列将会运行很多次，直到用户关闭浏览器（或 `AppletViewer`），`destroy()`方法被调用。

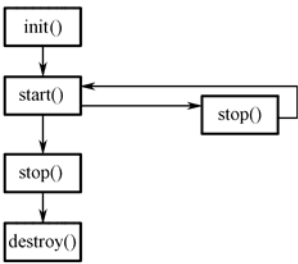


图 3-1 Applet 生命周期图

为了在行动中演示 Applet 生命周期方法，可以编写一个 Applet，其覆盖四种方法，然后计数各个方法执行的次数。

(1) 在文本编辑器打开一个新文本文件，输入如下 Applet 需要的导入语句：

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
```

(2) 输入如下的 LifeCycle Applet 程序头。该 Applet 中将包含用户单击的按钮，故将执行 (implement) ActionListener。

```
public class lifecycle
extends applet implements actionlistener;
```

(3) 按 Enter 键，为 class 输入一个开始花括号，然后再次按 Enter 键开始一个新行。

(4) 声明以下的六个标签对象，在 Applet 的生命周期内，将用来显示要执行的六种方法的名称。

```
Label messageInit = new label("init");
Label messageStart = new label("start");
Label messageDisplay = new label("display");
Label messageAction = new label("action");
Label messageStop = new label("stop");
Label messageDestroy = new label("destroy");
```

(5) 输入如下语句用以声明一个按钮：

```
button pressbutton = new button("press");
```

(6) 通过输入如下代码声明六个整型变量，它们分别用来装入各方法出现的次数：

```
int counInit,countStart,countDisplay,countAction,countStop,countDestroy;
```

(7) 添加如下的 init()方法，使 countInit 自动加 1，置组件于 Applet 中，然后调用 display()方法。

```
public void init()
{
    ++countInit;
    add(messageInit);
    add(messageStart);
    add(messageDisplay);
    add(messageAction);
    add(messageStop);
    add(messageDestroy);
    add(pressButton);
    pressButton.addActionListener(this);
    display();
}
```

(8) 添加如下的 start()方法，使 countStart 自动加 1，然后显示出来。

```
public void start()
{
    ++countStart;
    display();
}
```



```
}
```

(9) 添加如下的方法，使 `countDisplay` 自动加 1，然后显示六个方法的名称和其当前计数，指出方法执行的次数。

```
public void start()
{
    ++countDisplay;
    messageInit.setText("init"+ countInit);
    messageStart.setText("start"+ countStart);
    messageDisplay.setText("display"+ countDisplay);
    messageAction.setText("action"+ countAction);
    messageStop.setText("stop"+ countStop);
    messageDestroy.setText("destroy"+ countDestroy);
}
```

(10) 添加如下所示的 `stop()` 和 `destroy()` 方法，各自的计数器自动加 1，且调用显示方法。

```
public void stop()
{
    ++countStop;
    display();
}

public void destroy()
{
    ++countDestroy();
    display();
}
```

(11) 当用户单击按钮时，开始执行下面的 `actionDerformed()` 方法，使 `countAction` 自动加 1 并且显示出来。

```
public void acionDerformed(ActionEvent e)
{
    ++countAction;
    display();
}
```

(12) 为此 `class` 输入结束花括号。将此文件以 `lifeCycle.java` 为名字保存在磁盘中，如果有需要，可以编译、改错并再次编译。

下面，花一点时间来检查在 `LifeCycle.java` 中建立的代码。六个计数器分别在各自的方法中都会自动加 1，但不能显性地调用这些方法（除了 `display()` 方法）；其他的方法将会自动调用。接下来，将创建一个 `HIML` 文档来测试 `LifeCycle.java`。

为创建一个 `HTML` 文档去测试，应执行以下操作：

① 在文本编辑器，打开一个新文本文件。

② 输入如下 `HTML` 文档：

```
<html>
<applet code="LifeCycle.class"width=460 height=200>
</applet>
</html>
```

③ 将此文件以 `life.html` 为名称保存在磁盘中。

④ 用 `AppletViewer life.html` 命令运行该 HTML 文件，显示输出结果为：“init 1 start 1 display 2 action() stop() destroy ()”。当 Applet 运行时，`init()`方法被调用，所以 `countInit` 自动加 1，`init()`方法调用 `display()`，所以 `countDisplay` 自动加 1。在 `init()`执行后，`start()`紧接着执行，`countStart` 自动加 1，`start()`调用 `display()`，`countDisplay()`又自动加 1，所以第一次看到该 Applet 的 `countInit` 为 1，`countStart` 为 1，`countDisplay` 为 2。这时，`actionPerformed()`与 `stop()`、`destroy()`方法还没被执行。

⑤ 单击最小化按钮将 AppletViewer 窗口最小化，然后恢复它。此刻 Applet 的 `init()`方法仍然被调用了一次，且当最小化该 Applet 时，`start()`与 `stop()`方法都调用了 `display()`，所以 `countDisplay()`变了两次，目前的值为 4。

⑥ 将窗口又一次最小化然后最大化。现在结果为：`stop()`方法执行了两次，`start()`方法执行了 3 次，`display()`方法共执行了 6 次。

⑦ 单击“press”按钮，此时 `actionPerformed()`为 1，并且调用 `display()`，所以 `countDisplay` 增加为 7。

⑧ 继续最小化、最大化窗口，然后单击“press”按钮，记录每次产生的变化，直到能准确预测输出的结果。

注意：`destroy()`方法直到关闭 Applet 时才被执行。

3.2.2 一个全交互的Applet

现在已能创建一个相对复杂的应用程序或 Applet。接下来，将创建另一个 Applet，其中包括一些组件，接收用户输入，做出决定，使用数组完成输出，对 Applet 生命周期做出反应。

下面的 PartyPlanner Applet 是由 Event Handlers Incorporated 公司预估晚会价格的程序。该公司使用了一种合理的计算经费的算法，即其价格曲线为：随着邀请客人总数的增多，每位客人消费会减少。表 3-1 显示了价格结构。

表 3-1 晚会的每位客人的价格

客人数	每位客人费用
1~24	\$27
25~49	\$24
50~99	\$22
100~199	\$19
200~499	\$17
500~999	\$14
1000 及以上	\$11

此 Applet 程序让用户输入参加客人总数，用户可按 `Enter` 键或者单击鼠标去执行程序，对晚会价格进行计算，则 Applet 会立即显示每位客人的费用以及晚会的总花销。用户可输入客人总数，查看客人单价等信息，并确保下次启动程序时，客人数清为零。

为开始创建一个交互式的聚会计划 Applet，执行以下操作：

- (1) 在文本编辑器中打开一个新文本文件。
- (2) 输入如下导入语句、PartyPlanner 类的程序头，以及此 class 的开始花括号：

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
```

```
public class PartyPlanner
extends Applet implements ActionListener
{
```

(3) 程序中需要用到一些组件：一个用于公司名称的标签，用户可以单击进行运算的按钮，输入客人人数，一个用来让用户输入来访客人数的文本框，以及两个标签用来显示输出。输入如下代码来实现以上组件。

```
Label companyName=new label("Event HandlersIncorporated");
Button calcButton=new button("Calculate");
Label prompt=new label("Enter the number of guests at your event");
TextField numGuests=new TextField(5);
Label perPersonresult=new Label("plan with us. ");
Label totalResult=new Label("he more the merrier! ");
```

(4) 另外，为了使程序更具备可观性，可以输入如下语句来创建字体：

```
Font bigFont=new Font("Helvetica", Font.italic, 24);
```

(5) 通过输入如下语句，可以使用 `init()` 方法将组件添加到 Applet 中，并且准备好按钮以及能接收行动消息的文本框。

```
Public void init()
{
companyName.setFont(bigFont);
add(companyName);
add(prompt);
add(numGuests);
add(calcButton);
calcButton.addActionListener(this);
numGuests.addActionListener(this);
add(perPersonResult);
add(totalResult);
}
```

(6) 添加如下的 `start()` 方法，当用户离开 Applet、重置结果标签及数据输入文本框时执行该方法。

```
{
perPersonResult.setText("Plan with us. ");
numGuests.setText("0");
totalResult.setText("The more the merrier! ");
invalidate();
validate();
}
```

(7) 将此部分完成的 Applet 以 `PartyPlanner.java` 为名字保存在磁盘中。

现在已为 `PartyPlanner Applet` 完成了 `init()` 和 `start()` 方法，将各组件放置于 Applet 中，并在用户每次离开又返回 Applet 时重新初始化各组件。目前为止，Applet 并没有实际做任何事情，大多数 Applet 的工作包括在 `actionPerformed()` 方法中，它是此 Applet 中最复杂的方法。下面将创建 `actionPerformed()` 方法。它定义两个平行的数组：一个数组装对应 6 个比例的客人人数；另一个数组装这 6 个比率。

为完成 PartyPlanner Applet 应执行以下操作。

① 为 actionPerformed()输入如下程序头，并声明两个 guest 及 rates 数组。

```
public void actionPerformed(ActionEvent e)
{
    int[] guestLimit={0,25,50,100,200,500,1000};
    int[] ratePerGuest={27,25,22,19,17,14,11};
```

② 输入如下变量来存放客人人数。用户将向文本框输入信息，但是需要设置一个整型变量来完成计算，故可以使用 parseInt()方法。

```
int guests=integer.parseInt(numGuests.getText());
```

③ 需要两个变量，一个存放每人所花的费用，另一个存放全部聚会的费用。输入如下变量：

```
int individualfee=0,eventfee=0;
```

④ 输入如下变量作为数组下标：

```
int x=0,a=0;
```

有许多方法在 ratePerGuest 数组中查找每个人经费合适的位置。一种可能是使用 for 循环，下标从 5 变化到 0。如果客人的数目大于或等于任何在 guestLimit 数组的值，则在 ratePerGuest 数组中相应的每个人的费用率是正确的比率，在找到单个人的费用率后，可以乘以总人数得到整个晚会的费用。寻到正确的单个人费用后，不用再循环查找，故置下标 x 为 0，强迫其跳出循环。

⑤ 输入如下的 for 循环：

```
for(x=5;x>=0;--x)
if(guests>=guestLimit[x])
{individualFee=ratePerGuest[x];
 eventFee=guests*individualFee;
 x=0;
}
```

⑥ 在 actionPerformed()方法中，唯一的任务是为用户生成输出。输入如下代码来完成这个过程：

```
perPersonResult.setText
("$"+individualFee+"per person");
totalResult.setText("Event cost$"+eventFee);
```

⑦ 输入两个结束花括号，一个用来终止 actionPerformed()方法，另一个表示整个 PartyPlanner Applet 的结束。

⑧ 保存该文件，然后在命令提示符将其编译。

⑨ 在文本编辑器打开新文本文件，然后创建如下的 HTML 文件来测试该 Applet：

```
<HTML>
<APPLET CODE="PartyPlanner.class"WIDTH=320 HEIGHT=200>
</APPLET>
</HTML>
```

⑩ 将此 HTML 文件以 PartyPlan.html 为名字保存在磁盘里，然后使用 AppletViewer 命令来运行该文件。所生成的输入见表 3-1，用不同的客人人数来测试这个 Applet，直到确信单个人费用和晚会费用是正确的。最小化窗口并还原窗口来观察计算的费用。

⑪ 关闭 AppletViewer 窗口。

3.2.3 使用setLocation()方法

迄今为止上述所编写的一系列对象的缺点是不能选择放置在 Applet 中的标签和按钮对象的位置。当使用 add()方法向 Applet 当中添加组件时，它放在默认的物理位置，但可以使用 setLocation()方法来改变组件的位置。

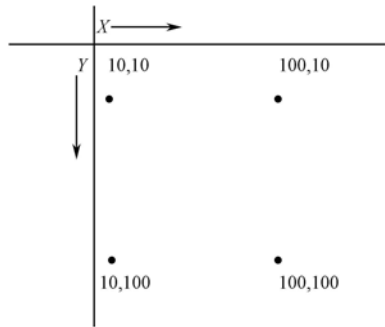


图 3-2 坐标位置图

setLocation()方法允许将组件放置在 AppletViewer 窗口中的任一特殊位置。任何 Applet 窗口都是由屏幕上的水平的和垂直的像素所组成的，可以在 HTML 文档中设置像素的值来测量 Applet。任何设置在 Applet 中的组件都有一个水平的 X 坐标轴和垂直的 Y 坐标轴。在左上角显示的位置为 0, 0。第一个数也就是 X 值，从左到右数值逐渐变大，Y 值从上到下逐渐增加，如图 3-2 所示，绘出坐标位置。

例如，为了将一个名为 someLabel 的标签对象的位置定在窗口左上角，可以设置为“someLabel.setLocation(10,10);”。如果一个窗口的高为 100 像素，宽为 200 像素，可以输入语句“pressMe.setLocation(100,50);”，将一个名为 Press Me 的按钮设置在坐标中心的位置上。

下面将建立一个通过单击 Press Me 按钮改变其位置的标签。为建立一个可以移动的标签，应执行下面的操作。

(1) 在文本编辑器打开一个新文本文件，然后输入下面需要的导入语句：

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
```

(2) 输入名为 MoveLabel 的 class 的开头部分。

```
Public class MoveLabel
    Extends Applet implements ActionListener
{
```

(3) 定义下面的标签、按钮和两个分别用来存放标签水平和垂直坐标的整型变量。

```
Label movingMsg=new Label("Event Handlers Inc. ");
Button pressButton=new Button("Press");
Int xLoc=20,yLoc=20;
```

(4) 输入下面的 init()方法来添加组件到 Applet 屏幕，并且准备一个接收消息的按钮。

```
public void init()
{
    add(movingMsg);
    add(pressButton);
    pressButton.addActionListener(this);
}
```

(5) 当用户单击 Press Me 按钮时，消息向右移动 10 个像素，向下移动 10 个像素。换句话说，它将向屏幕的右下方运动一个角度。输入下面的 actionPerformed()方法：

```
public void actionPerformed(ActionEvent e)
{
```

```
    moveingMsg.setLocation(xLoc+=10,yLoc+=10);  
}
```

(6) 对 class 添加结束花括号。

(7) 文件以 MoveLabel.java 作为名字存储在磁盘上。

(8) 在文本编辑器中打开一个新文件，然后创建下面的 HTML 文档来测试 Applet。

```
<HTML>  
<APPLET CODE="MoveLabel.class"WIDTH=460 HEIGHT=300>  
</APPLET>  
</HTML>
```

(9) 将此 HTML 文档以 move.html 作为文件名，存储到磁盘。然后，在命令提示符下，使用 AppletViewer 命令运行该文件。观察单击 Press Me 按钮时，标签如何移动。

(10) 关闭 AppletViewer 窗口。

3.2.4 使用setEnabled()方法

有时组件中计算机程序用户不能运行，例如，当程序员不想让用户去访问某按钮的功能而使它变暗时，可以对一个组件使用 setEnabled()方法来使其失效，并且当想要使组件有效时又能使其恢复。setEnabled()方法可以设置变量的值为 true 或 false，从而使组件恢复效用或失效。

当创建一个组件时，它的默认值为“可行的”。例如，在 MoveLabel Applet 中，用户可以不断单击按钮从而使标签移动到完全从屏幕中消失。如果想阻止这种情况，可以使标签在移动一段想移动的距离后失效。下面是标签达到离 Y 轴 280 像素处停止的程序。

为使按钮失效，应执行下面的操作步骤。

(1) 在文本编辑器，打开 MoveLabel.java 文件。

(2) 置光标于 actionPerformed()方法的结尾，并按 Enter 键来开始新文本行。然后添加如下语句，当消息移动到 Y 轴 280 像素时，使按钮失效。

```
    if(yLoc==280)  
        pressButton.setEnabled(false);
```

(3) 存储程序、编译，并用 AppletViewer 命令运行 move.html 文件。单击按钮直到按钮失效，并且标签不能起任何作用。

(4) 关闭 AppletViewer 窗口，并且关闭文本编辑器。

3.2.5 得到帮助

至此我们已编制了较为复杂的程序，包括若干方法和许多独立的程序。随着继续学习 Java 和 Applet 编程，将会很容易地编写比现在复杂数十倍的程序。有许多方法可在如下网址得到帮助：<http://Java.sun.com>。下面列举了一些基于 Web 的 Java 新闻组：

```
comp.lang.java.advocacy  
comp.lang.java.announce  
    comp.lang.java.api  
    comp.lang.java.beans  
    comp.lang.java.gui  
    comp.lang.java.help
```

```
comp.lang.java.misc  
comp.lang.java.programmer  
comp.lang.java.security  
comp.lang.java.setup  
comp.lang.java.softwaretools  
comp.lang.java.tech
```

3.3 JDBC及其应用

3.3.1 JDBC编程技术

1. ODBC到JDBC的发展

与 JDBC（Java 数据库连接）有关的一个众所皆知的词语是开放式数据库互连（Open Database Connectivity, ODBC），它是用 C 语言实现的标准应用程序数据库接口，用来在数据库管理系统(DBMS)中存取数据。通过 ODBC API，应用程序可以存取保存在多种不同 DBMS 中的数据，不论这些 DBMS 使用的是何种数据存储格式和编程接口。

ODBC 接口支持 SQL 语句的使用，这使得 ODBC 将其应用系统的范围扩展到众多的数据库环境中。另外，微软所有主要的应用程序和开发环境都支持 ODBC，所以，ODBC 事实上成为一种工业标准。

自从 Java 语言诞生起就得到了广泛的应用，出现了大量的用 Java 语言编写的程序，当然也包括数据库应用程序。但最初并没有一个 Java 语言的 API，编程人员不得不在 Java 程序中加入 C 语言的 ODBC 函数调用，这样就大大影响了 Java 特性的发挥。随着 Java 的日益普及，对 Java 语言接口中访问数据库 API 的要求越来越强烈。为此，Sun 公司开发了以 Java 语言为接口的数据库应用程序开发接口——JDBC。

2. JDBC技术概述

Java 数据库连接（Java Database Connectivity, JDBC）是一种用于执行 SQL 语句的 Java API（应用程序设计接口）。它由一组用 Java 语言编写的类和接口组成，可以为多种关系数据库提供统一访问。JDBC 为数据库应用开发人员和前台工具开发人员提供了一种标准的应用程序设计接口，使开发人员可以用纯 Java 语言编写完整的数据库应用程序。

通过使用 JDBC，开发人员可以很方便地将 SQL 语句传送给几乎任何一种数据库。也就是说，开发人员可以不必写一个程序访问 Sybase，然后写另一个程序访问 Oracle，再写一个程序访问 Microsoft 的 SQL Server。而是使用 JDBC 写的程序自动地将 SQL 语句传送给相应的数据库管理系统。另外，由于 Java 具有平台无关性的特性，所以 Java 和 JDBC 的结合可以让开发人员在开发数据库应用时真正实现“Write Once, Run Anywhere!”。

JDBC 使得 Java 的应用能够同各种各样的数据库连接起来，因此扩展了 Java 的能力。Java 的一个突出特性就是支持 Applet（小应用程序）在 Web 中的应用，所以这种扩展性主要表现在对 Web 中数据库的访问技术上。例如，使用 Java 和 JDBC API 可以公布一个 Web 页，页中包含能够访问远端数据库的 Applet。这样就方便了人们对网上不同类型数据库资源

的访问，实现了网络资源的共享。

3. JDBC的用途

JDBC 是底层 API，它可以直接调用 SQL 语句。同时它也是构造高级 API 和数据库开发工具的基础。大致来讲，JDBC 主要有三方面的用途：

- (1) 与数据库建立连接。
- (2) 向数据库发送 SQL 语句。
- (3) 处理数据库操作返回的结果。

例 3-1 的代码是使用 JDBC 的一个基本例子，其中包含了以上三种用途。

【例 3-1】使用 JDBC 的一个例子。

```
Connection con=Drivermanager.getConnection("jdbc:
    odbc:example","login","password");
//与指定的数据库 example 建立连接
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("SELECT a, b, c FROM Table1");
//显示查询结果
While (rs.next()){
    Int x=rs.getInt("a");
    String s=rs.getString("b");
    Float f=rs.getFloat("c");
}
```

通过前面的了解，可以描述出 JDBC 的工作模型。

在图 3-3 的两层模型中，用户的 SQL 语句传送给数据库，而这些语句执行的结果将被传回用户。数据库可以在同一机器上，也可在另一机器上通过网络进行连接。在这种结构中，用户的计算机作为 Client（客户），运行数据库的计算机作为 Server（服务器）。

在图 3-4 的三层模型中，命令将发送到服务的“中间层”，而中间层将 SQL 语句发送到数据库。数据库处理 SQL 语句并将结果返回“中间层”，然后“中间层”将它们返回用户。因为中间层可以对访问进行控制并辅助数据库更新，另外可以将用户易用的高层的 API 转换成底层 API 调用，所以三层模型在实际应用中可以提供更好的性能。

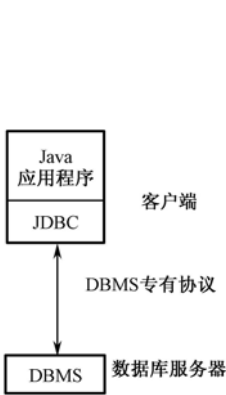


图 3-3 JDBC 的两层模型

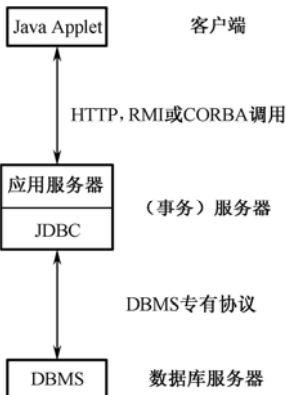


图 3-4 JDBC 的三层模型

3.3.2 使用JDBC访问数据库

用户想要实现对数据库的访问，一般要经过如下步骤：建立 JDBC-ODBC 桥接器→连接到数据库→向数据库发送 SQL 语句→处理查询结果。在了解这些步骤之前，首先要对 JDBC API 的结构和使用有所了解。图 3-5 是 JDBC API 的结构。

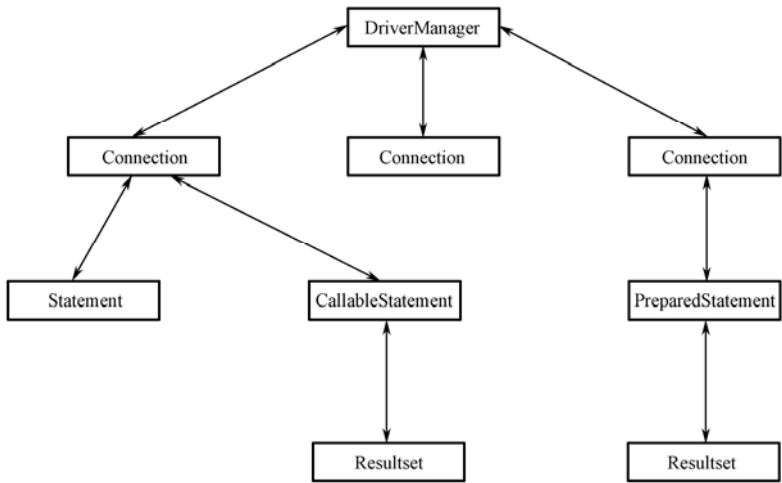


图 3-5 JDBC API 的结构

JDBC 的类和接口为：

- java.sql.Connection——完成对某一个指定数据库连接的功能。
- java.sql.Statement——在一个给定的连接中作为 SQL 语句执行的容器。
- java.sql.PrepareStatement——用于执行预编译的 SQL 语句。
- java.sql.CallableStatement——用于返回执行数据库中存储过程调用。
- java.sql.ResultSet——查询语句返回的结果集。

下面详细介绍如何使用 JDBC 来访问数据库。

1. 使用JDBC与数据库建立连接

1) 使用 JDBC-ODBC 桥

为了连接到某个数据库，首先要建立一个 JDBC-ODBC 桥接器。JDBC-ODBC 桥是一个 JDBC 驱动程序，它通过将 JDBC 操作转换为 ODBC 操作来实现 JDBC 操作。JDBC-ODBC 桥为所有对 ODBC 可用的数据库实现 JDBC，它作为 sun.jdbc.odbc 包实现，其中包含一个用来访问 ODBC 的本地库。因为 ODBC 被广泛地使用，该桥的优点是让 JDBC 能够访问几乎所有的数据库。那么如何使用 JDBC-ODBC 桥呢？这里介绍用 Java 类加载器将其显示加载的方法：

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Class 是包 java.sql 中的一个类，该类通过调用它的静态方法 forName() 就可以建立 JDBC-ODBC 桥接器。值得注意的是，在实际应用中，建立桥接器时可能发生异常，因此必须捕获这个异常，所以建立桥接器的标准方法是：

```
try {Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); }
catch (ClassNotFoundException e) { }
```

2) 建立一个连接

建立了 JDBC-ODBC 桥接器后, 就可以与指定的数据库建立连接了。首先使用 `java.sql` 中的 `Connection` 类声明一个对象, 再使用类 `DriverManager` 调用它的静态方法 `getConnection()` 创建这个连接对象:

```
Connection con=DriverManager.getConnection("jdbc:odbc:数据源名字", "数据源的注册用户名", "数据源的口令");
```

其中后两个字符串可以是空串。

例如, 为了和数据源 `jxgl` (注册用户名为 `teacher`, 数据源的口令是 `flower`) 建立连接, 创建 `Connection` 对象的方法如下:

```
Connection con = DriverManager.getConnection("jdbc:odbc:jxgl", "teacher", "flower");
```

当然, 为了在建立连接时能够捕获异常, 标准的方法是:

```
try{
Connection con=DriverManager.getConnection("jdbc:odbc:jxgl", "teacher",
    "flower");
}
catch (SQLException e) {}
```

连接的断开可以使用方法: `close()`。

3) JDBC-ODBC 桥支持的 JDBC URL

在前面已接触过统一资源定位器 (Uniform Resource Locator, URL) 的概念, 它能够在 Internet 上定位资源的有关信息。JDBC URL 提供了一种标识数据库的方法, 它可以使相应的驱动程序能够识别该数据库并与之建立连接。

JDBC URL 的标准语法由三部分组成, 各部分之间用冒号分隔。

`jdbc:<子协议>:<子名称>`

(1) `jdbc` 表示协议。JDBC URL 中的协议总以 “`jdbc`” 开头。

(2) `<子协议>` 表示驱动程序名或数据连接机制名称。子协议名的典型示例是 “`odbc`”, 该名称是为用于指定 ODBC 风格的数据资源名称的 URL 专门保留的。

(3) `<子名称>` 是数据库的标识。使用子名称的目的是为定位数据库提供足够的信息。前面的例子中数据库位于本地机上, 所以子名称就是数据库的名称。如果要访问 Internet 中的数据库, 子名称就必须将数据库所在的网络地址包括进去, 而且必须遵循标准的 URL 命名约定:

//主机号: 端口/子名称

例如, 要访问一个位于主机号为 `dataserver`, 端口为 `8080`, 子名称为 `datasource` 的数据库资源, JDBC-ODBC 桥支持的 JDBC URL 应为:

```
jdbc:odbc://dataserver:8080/datasource
```

相应的与此数据库建立连接的代码为:

```
String url = "jdbc:odbc://dataserver:8080/datasource"
try {Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); }
catch (ClassNotFoundException e) {}
try {
Connection con=DriverManager.getConnection(url, "teacher", "flower");
}
```

```
catch (SQLException e) {}
```

2. JDBC的查询发送

JDBC 提供了三种对象来实现查询语句的发送执行。它们是 `Statement` 对象，`PreparedStatement` 对象和 `CallableStatement` 对象。它们都是可以执行 SQL 语句的容器对象。`CallableStatement` 继承了 `PreparedStatement`，而 `PreparedStatement` 继承了 `Statement`。它们用于执行不同类型的 SQL 语句：`Statement` 对象用于执行简单的不含参数的 SQL 语句；`PreparedStatement` 对象用于执行带有 IN 类型参数的预编译过的 SQL 语句；而 `CallableStatement` 对象用于执行数据库的存储过程。

1) 创建和使用 `Statement` 对象

(1) 某个数据库建立连接后，就可以创建一个 `Statement` 对象，用来发送 SQL 语句（不含参数），并得到 SQL 语句执行的结果。创建一个 `Statement` 对象可以通过 `connection` 类方法 `createStatement()` 来实现，具体方法如下：

```
Connection con = DriverManager.getConnection(url, "userID", "pwd");  
Statement stmt = con.createStatement();
```

(2) 创建一个 `Statement` 对象后，就可以使用这个 `Statement` 对象。`Statement` 对象提供的常用方法有以下几种。

① `ResultSet executeQuery(String sql) throws SQLException`：该方法执行一个 SQL 语句，返回一个 `ResultSet` 类型的结果。例如：

```
Connection con = DriverManager.getConnection(url, "userID", "pwd");  
Statement stmt = con.createStatement();  
ResultSet rs = stmt.executeQuery("SELECT column1, column2, column3 FROM  
    jxgltable");
```

在这里只提供了简单的实现语句，在实际应用中，因为要判断异常情况，所以这部分介绍的所有方法的使用均应做异常处理。这里就不再一一描述。

② `int executeUpdate("String sql")throws SQLException`：执行一个 SQL INSERT，UPDATE 或 DELETE 语句，并返回相应操作成功的记录数，所以该方法的执行结果类型为整型。例如：

```
stmt.executeUpdate("DROP TABLE jxgltable1");//删除表 jxgltable1
```

③ `boolean execute(String sql)throws SQLException`：执行一个可能会返回多个结果的 SQL 语句。如果结果为一个 `ResultSet`，返回值为 `true`。

④ `void close()throws SQLException`：释放 `Statement` 对象和 JDBC 资源。例如：

```
rs.close() //关闭结果集  
stmt.close() //关闭 Statement 对象
```

2) 创建和使用 `PreparedStatement` 对象

`PreparedStatement` 对象继承了 `Statement` 对象的所有功能，它与 `Statement` 对象存在两方面的不同。

(1) `PreparedStatement` 对象代表一个已经编译过的 SQL 语句，就是语句“准备好”的意思。

(2) 在 `PreparedStatement` 对象中的 SQL 语句可以包含一个或多个 IN 参数。IN 参数就是指那些在 SQL 语句创立时尚未指定具体值的参数。每个 IN 参数的位置上用一个“?”来

代替。当然 IN 参数的具体值必须在执行之前确定下来。

由于 PreparedStatement 对象已预编译过，所以其执行速度要快于 Statement 对象。对于需要经常多次执行的 SQL 语句，通常为其创建 PreparedStatement 对象，以提高效率。PreparedStatement 对象提供了一整套方法，用来设置 IN 参数的具体值。例如，下面代码实现了 PreparedStatement 对象的创建，其中包含两个 IN 参数，并通过一定的方法给参数赋值：

```
Connection con=DriverManager.getConnection(url, "userID", "pwd");
PreparedStatement pstmt=con.prepareStatement("UPDATE jxgltable SET
    grade=? WHERE xm=?");
Pstmt.setFloat(1,90.0);
Pstmt.setString (2, "刘炎");
```

通过上面两个方法的调用，完成了对参数赋值的操作。即设置第一个“?”的值为“刘炎”，从而实现将指定表 jxgltable 中 xm 字段为“刘炎”的记录的 grade 字段内容修改为 90.0。所有为参数赋值的 setXXX 方法都遵循以下规则：方法名中的 XXX 是与该参数相应的类型名，例如，如果参数具有 Java 类型 long，则使用的方法就是 setLong；方法中的第一个参数 parameterIndex 为要设置的参数的序位号；第二个参数是设置给该参数的值，length 为流中字节数。下面是 PreparedStatement 对象提供的对参数赋值的方法：

```
void setNull(int parameterIndex, int sqlType) throws SQLException
    //设置参数为默认类型
void setBoolean(int parameterIndex, boolean x) throws SQLException
    //设置参数为布尔类型
void setByte(int parameterIndex, byte x) throws SQLException
    //设置参数为字节类型
```

Statement 提供的三种常用方法 executeQuery, executeUpdate, execute 被继承下来，但有变动，即其中的参数被去掉。也就是说，这三种被 PreparedStatement 对象调用时均是无参数的方法。例如，续上段代码：

```
PreparedStatement pstmt=con.prepareStatement("UPDATE jxgltable SET
    grade=? WHERE xm=?");
Pstmt.setFloat(1,90.0)
Pstmt.setString (2, "刘炎");
Pstmt.executeUpdate(); //返回相应操作成功的记录数
```

3) CallableStatement 对象

CallableStatement 对象继承了前面两种类提供的方法，CallableStatement 对象提供了一种调用数据库存储过程的方法。存储过程存储在数据库端，调用存储过程则包含在 CallableStatement 对象中。通常使用 CallableStatement 对象必须首先要知道 DBMS 是否支持存储过程，并且支持什么样的过程。

3. JDBC的结果接收

JDBC 的结果接收是通过 ResultSet 对象来实现的，它通常是 Statement 对象执行而产生的结果。一个 ResultSet 对象包含了执行某个 SQL 语句后满足条件的所有行。此外，它还提供了对这些行的访问，用户可以通过一组 get 方法来访问当前行的不同列。通过 ResultSet.next 方法可以取到 ResultSet 的下一行。

```
Boolean next() throws SQLException //当前行后如果没有记录，该方法返回值 false
```

可以将结果看成是一张二维表格，表头是查询字段，表中的记录就是查询返回的结果。例如，查询语句是 `SELECT column1, column2, column3 FROM jxgltable`，可能的结果集如表 3-2 所示。

表 3-2 查询记录结果集

column1	column2	column3
Stud01	Janny	80
Stud02	Make	90
Stud03	Joneli	90

(1) 结果集的行和行指针。每个结果都包含一个指针来指示当前行。当 `next()`方法被调用后，指针向下移动了一行。指针的初始位置指向第一行之前，所以第一次调用 `next()`方法时，指针才指向结果集中的第一行。通过移动结果集中的行指针，使得用户可以浏览整个结果集中的内容。在所使用的 DBMS 支持相应的更新、删除等功能时，还可以对结果集中的相应内容进行修改。

(2) 结果集中的列。`ResultSet` 类提供了 `getXXX` 方法允许用户从当前行中获取列值。`getXXX` 方法可以根据字段名或字段的索引值来获取列的内容。例如，某个结果集中第 2 列的列名是“column2”，相应的字段的类型是 `string` 类型，那么可以使用两种方法来获取这一列的值：

```
String s=rs.getString("column2")
String s=rs.getString(2);
```

一般来说，使用字段的索引效率更高。字段索引的编号从 1 开始，1 为当前记录的第一个字段，2 为第二个字段，依次类推。这里就不再一一列举所有的 `getXXX` 方法了。注意，`PreparedStatement` 对象在参照 `setXXX` 方法时，方法中的第一个参数在这里可以有两种表示方法：`columnIndex`（列的位置索引，从 1 开始），`columnName`（使用列名）。

(3) 结果集内容的显示。利用循环结构编写程序来显示结果：

```
rs=stmt.executeQuery("SELECT * from DemoTable ORDER BY test_id");
//查询数据库中的表 Demotable，得到以 test_id 排序后的所有记录，并存储到 ResultSet rs
//对象中
System.out.println("Result of Query: ");
While (rs.next())
{
    int theInt=rs.getInt("test_id");
    String str=rs.getString("test_val");
    System.out.println("\ttest_id="+theInt+"\tetetr="+str);
}
```

(4) 结果集中的数据类型和转换。对于 `getXXX` 方法来说，JDBC 进行了从基本的 SQL 类型到 Java 类型的转换，并返回一个合适的 Java 值。例如，`getString()`方法要获取 `VARCHAR` SQL 类型的数据，JDBC 数据库驱动器将 `VARCHAR` 转换成 Java 的 `String` 类型，返回值将是 Java 的 `String` 对象。

(5) 对非常大的行值可以采用流的方法。`ResultSet` 对象可以接收任意长的 `LONGVARBINARY` 和 `LONGVARCHAR` 数据。`GetByte` 方法和 `getString` 方法将数据以一个大数据包的形式返回。可以利用一个更方便的方法来将数据以更小的，大小固定的数据包返

回，这种方法就是让 `ResultSet` 类返回 `java.io.input` 流来实现。JDBC API 提供三种方法来得到不同类型数据的流：

`getBinaryStream()`：返回一个流，数据类型不做任何转换；

`getAsciiStream()`：返回单字节的 ASCII 字符流；

`getUnicodeStream()`：返回双字节的 unicode 流。

通过使用这些流，用户就可以将非常大的行值以流的方法进行处理。

3.3.3 应用实例

假设已经在 ACCESS 环境下创建了 `guestbase.mdb` 数据库，其中有 `yhkl` 表，表的结构包含用户名、口令字段。下面三个例子完成对表的查询、修改、添加等操作。

1. 数据库中表的内容

【例 3-2】 用户口令的查询。

```
import java.awt.*;
import java.net.*; //对数据库进行的操作必须安装如下类包
import java.sql.*;
import java.io.*;
import java.lang.*;

public class datawindow extends Frame {
    Button button ;
    TextField text;
    Connection con;
    Statement stmt;
    String kl,yhm;
    datawindow() {
        super("数据库查询演示");
        setVisible(true);setLayout(new FlowLayout());
        button=new Button("ok");
        Text=new TextField(8);
        add(button);add(text);
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Con=DriverManager.getConnection("jdbc:odbc:guestbase", "lxl", "wyw");
            //与数据库建立连接
            stmt=con.createStatement();
            string s="`"+text.getText().trim()+"`";
            resultSet rs=stmt.executeQuery("SELECT * From yhkltable WHERE 用户名= "+s);
            //将查询结果放入结果集中
            While (rs.next()) { //循环输出结果集中的内容
                yhm=rs.getString ("用户名");
                kl=rs.getString("口令");
                system.out.println("\t 用户名="+ymh+"\t 口令="+kl);
            }
        }
```

```

        con.close();
        text.setText(yl);
    }
    catch(java.lang.Exception ee){} //捕获异常
}
}

```

【例 3-3】 结合线程实现实时查询，从而得到最新的数据。本例的功能是每隔 2 s 更新一次显示，所以如果数据库内容被更新，用户便可以查询到最新的内容。

```

class dataaccess extends Thread{
JFrame myframe=new JFrame("实时查询演示");
JTextArea text=new JTextArea();
String yhm,yl;
Connection con;
Statement stm;
ResultSet rs;
dataaccess(){
    myframe.setBounds(100,100,300,300);
    myframe.setVisible(true);
    myframe.setLayout(new BorderLayout());
    myframe.add(text, "Center");
    myframe.pack();
    myframe.addWindowListener(new WindowAdapter(){
        public void windowclosing(WindowEvent e){
            system.exit(0);}
    });
    try{
        Class.forName("sun .jdbc.odbc.JdbcOdbcDriver");
    }
    catch(ClassNotFoundException e){}
    try{
        //建立连接
        con=DriverManager.getConnection("jdbc.odbc.guestbase", "lxl", "wyw");
        stm=con.createStatement();
    }
    catch(SQLException e){}
}
public void run(){
    //定义线程运行所要完成的工作
    while(true){
        text.setText(null);
        try{
            rs=stm.executeQuery("SELECT * FROM yhkltable");//查询
            while(rs.next()){
                yhm=rs.getString("用户名");
                yl=rs.getString("口令");
                text.append("用户名"+yhm+"\n");
                text.append("口令"+yl+"\n");
            }
        }
    }
}

```

```

    }
}
catch(SQLException e1){}
myframe.pack();
try{
    sleep(2000);
}
catch(InterruptedException exp){}
}
}
}
public class accesstest{
    public static void main(String args[]){
        dataaccess myaccess=new dataaccess();
        myaccess.start();
    }
}

```

2. 更新、添加数据库中表的记录

【例 3-4】上例完成的是数据库查询的功能，本例演示如何对用户口令表中记录的内容进行更新和添加。

```

public class modify extends JFrame implements ActionListener
{
    JTextField yhm,kl;
    JButton gx,tj;
    Connection com=null;
    Statement stm=null;
    modify()
    {
        Container cont=getContentPane();
        super("数据库修改操作演示");
        cont.setVisible(true);
        cont.setLayout (new GridLayout(2,1));
        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        }
        catch(ClassNotFoundException e){}
        try
        {
            con = DriverManager.getConnection("jdbc:odbc:guestbase", "lxl", "whw");
            stm=con.createStatement();
        }
        catch(SQLException ee){}
        gxym=new JTextField(8);
    }
}

```



```

tjyhm=new JTextField(8);
gxkl=new JTextField(6);
tjkl=new JTextField(6);
gx=new JButton("更新");
tj=new JButton("添加");
JPanel p1=new JPanel();JPanel p2=new JPanel();
p1.add(new JLabel("输入要更新的用户名"));
p1.add(gxyhm);
p1.add(new JLabel("输入要更新的口令"));
p1.add(gxkl);
p1.add(gx);
p2.add(new JLabel("输入要添加的用户名"));
p2.add(tjyhm);
p2.add(new JLabel("输入要添加的口令"));
p2.add(tjkl);
p2.add(tj);
cont.add(p1);
cont.add(p2);
ex.addActionListener(this);
tj.addActionListener(this);//添加监听器
addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent e)
    {
        system.exit(0);}
}
}
public void actionPerformed(ActionEvent e)
{
    if(e.getSource()==gx)
    {//单击“修改”按钮
        try
        {
            update();
        }
        catch(SQLException ee){}
    }
    else if(e.getSource()==tj)
    {//单击“添加”按钮
        try
        {
            append();
        }
        catch(SQLException e3e){}
    }
}

```

```

    }
    //定义修改方法
    public void update() throws SQLException
    {
        String s1="" +gxyhm.getText().trim()+"",
            s2="" +gxkl.getText().trim()+" ";
        con=DriverManager.getConnection("jdbc.odbc:guestbase","lxl","wyw");
        stm.extcuteUpdate("UPDATE yhkltable SET 口令="+s2+"WHERE 用户名
                            ="+s1+"");
        con.close();
    }
    //定义添加方法
    public void append() throws SQLException
    {
        String s1="" +tjyh.getText().trim()+"",
            s2="" +tjkl.getText().trim()+" ";
        con=DriVerManager.getConnection("jdbc.odbc:guestbase","lxl","wyw");
        stm.executeUpdate("INSERT INTO yhkltable VALUES("+s1+", "+s2+"");
        con.close();
    }
}
//定义主类创建上类的实例
public class dbmodify
{
    publi static void main(String args[])
    {
        modify user=new modify();
        user.modify();
    }
}

```

这里举的例子主要针对的操作环境是对本地机上数据库内容的访问，而应用在网页中的数据库访问工作是非常实用的，那么如何解决使用小应用程序在网页中来对数据库进行访问呢？实际上将本节所述的使用 JDBC 访问数据库的例子按 Applet 规范生成小应用程序，再用 HTML 调用此小应用程序即可。HTML 调用 Applet 本章前面已详述，此处不再赘述。

习 题 3

1. 选择题（I）

(1) Applet 是_____。

- | | |
|------------------|-----------------|
| A. 独立的程序 | B. 网页 |
| C. 从另外的应用程序中所调用的 | D. 用 HTML 语言编写的 |

(2) 下列那一项是对 AppletViewer 的正确描述？

- | | |
|------------------------|-------------------|
| A. 这是一种方法 | B. 必须自己编写它的代码 |
| C. 它是 Java 开发工具的自带配套工具 | D. 它必须从 HTML 文件调用 |

(3) HTML 是_____。

A. Hypertext Markup Language

B. Hash Table Management Language

C. Heap Task Monitoring List

D. How To Make a Lasting Impression

(4) 当编写一个 Java Applet 时, 要以_____为扩展名将其保存。

A. .app

B. .html

C. .java

D. .class

(5) Java 应用程序和 Java Applet 有相似之处是因为它们都_____。

A. 是用 javac 命令编译的

B. 是用 javac 命令执行的

C. 在 HTML 文档中执行

D. 都拥有一个 main()方法

(6) 为了在 HTML 文件中使用 Applet, 必须包括_____的名字。

A. .java 源代码文件

B. .class 编译文件

C. .exe 可执行文件

D. Applet 运行时所在的网页地址

(7) 允许在计算机显示器上显示 HTML 文档的程序是_____。

A. 搜索引擎

B. 编译器

C. 浏览器

D. 服务器

(8) HTML 的命令也可以称为_____。

A. 指令

B. 规则

C. 便签

D. 标记

(9) 所有的 HTML 命令都被_____所包围。

A. 圆括号

B. 花括号

C. 圆点

D. 尖括号

(10) 每一对 HTML 标签的结尾符前都被冠以_____。

A. 圆点

B. 前斜杠

C. 后斜杠

D. 分号

(11) 在 HTML 文档中调用使用 CODE 的任何 Applet 的名字要以_____为扩展名。

A. .exe

B. .code

C. .java

D. .class

(12) 通常, 应该在显示器端建立规格为_____像素的 Applet, 这样用户才能看到整个 Applet。

A. 100×100

B. 220×360

C. 640×480

D. 2200×100

(13) 标签和按钮都是_____。

A. 组件

B. 容器

C. Applet

D. 成份

(14) 将一个值置于以前构建的标签中所使用的方法是_____。

A. getValue()

B. setText()

C. fillLabel()

D. setValue()

(15) add()方法是_____。

A. 添加两个整数

B. 添加任何数据类型的两个数据

C. 将一个组件添加到 Applet Viewer 窗口中

D. 将一个文本值添加到一个 Applet 组件中

(16) 在每个 Applet 中不一定包含下列哪一种方法_____。

A. init()

B. add()

C. stop()

D. destroy()

(17) 被任何 Applet 第一个调用的方法是_____。

A. main()

B. start()

C. init()

D. 第一个在 Applet 中出现的方法

(18) Font 对象包含全部下列自变量, 除了_____。

A. language

B. typeface

C. style

D. point size

(19) 用户能向其中输入一整行文本信息的 Windows 组件是_____。

A. InputArea

B. DataFiled

C. TextFiled

D. Label

(20) Constructor Public Button(“4”)能创建_____。

- A. 一个无标签的按钮
- B. 一个 4 像素宽的按钮
- C. 一个 4 个字符宽的按钮
- D. 一个名称为 4 的按钮

(21) 可以使用_____方法改变按钮的标签。

- A. setText()
- B. getText()
- C. setLabel()
- D. getLabel()

(22) 用户可能会在_____程序中以任何顺序初始化任何数量的事件。

- A. 一个事件驱动
- B. 一个过程
- C. 一个随机
- D. 任何 Java

(23) ActionListener 是_____的一个例子。

- A. 导入
- B. Applet
- C. 接口
- D. 组件

(24) 当一个 Applet 用按钮登记为监听器，且用户单击时，执行的方法是_____。

- A. buttonPressed()
- B. addActionListener()
- C. start()
- D. actionPerformed()

(25) 当一个 Applet 过时时，用下列_____方法提出警告。

- A. date()
- B. change()
- C. invalidate()
- D. validate()

2. 建立一个带有标签为“Who is the greatest?”按钮的 Applet，当用户单击按钮时，显示出你的大号字的名字。

3. (1) 建立一个让用户输入密码到文本框然后按 Enter 键的 Applet。对比密码“Rosebud”，如果匹配成功，显示“欢迎访问”；如果不匹配，显示“拒绝访问”。

(2) 修改题(1)程序中的密码，使其忽略用户输入“Rosebud”密码时大小写字母的差异。

(3) 修改题(2)程序的密码，使其可接受如下合法的密码组：“Rosebud”、“Redrum”、“Jason”、“Surrender”，或者“Dorothy”。

4. 建立一个带有“今天是”标签的 Applet，而且当用户单击其按钮时，便可在一个文本框中显示日期和时间。

5. 建立一个 Applet，当用户在其一个文本框输入雇员姓名时，它可在另一个文本框显示雇员的职务名称，可以使用一个数组来存储雇员的姓名和职务名称。

6. 建立一个具有按钮的 Applet，以 8-point 字号显示你的姓名。每次用户单击按钮时，便为显示姓名增加 4 points 尺寸。当字号尺寸超过 24 points 时，去掉该按钮。

7. 选择题(II)

(1) 当写一个方法，如果它与自动提供的方法具有相同的方法头时，则_____原始版本。

- A. 破坏
- B. 覆盖
- C. 调用
- D. 复制

(2) 如果对 Applet 不写一个 init()方法，则_____。

- A. 你的程序将不编译
- B. 你的程序将编译，但不运行
- C. 你必须写一个 main()方法
- D. 一个自动提供的 init()方法执行

(3) 在 init()之后，立即执行的方法是_____。

- A. main()
- B. begin()
- C. start()
- D. stop()

(4) start()方法执行_____。

- A. 在 init()方法之后
- B. 每次 Applet 由不活动变成活动
- C. 以上两者都是
- D. 以上两者都不是

(5) 当用户离开页面时所执行的方法是_____。

- A. stop()
- B. destroy()
- C. kill()
- D. finish()

(6) 当用户_____时，destroy()方法被调用。

- A. 关闭浏览器或 AppletViewer 窗口
C. 离开页面

- B. 最小化 AppletViewer 窗口
D. 退出计算机

(7) stop()-start()序列_____在 Applet。

- A. 必定不会多于一次出现
C. 必定从未出现

- B. 可能出现任何次
D. 不经常出现

(8) 下面哪个语句创建了说“Welcome”的标签?

- A. Label=new Label("Welcome")
C. aLabel=new Label("Welcome")

- B. Label aLabel=Label("Welcome")
D. Label aLabel=new Label("Welcome")

(9) 下面哪语句正确地创建了一个 Font 对象?

- A. Font aFont=new Font("TimesRoman", Font.ITALIC, 20)
B. Font aFont=new Font(30, "Helvetica", Font.ITALIC)
C. Font aFont=new Font(Font.BOLD, "Helvetica", 24)
D. Font aFont=new Font(22, Font.BOLD, "TimesRoman")

(10) 将一个组件置位于 Applet 的方法是_____。

- A. position() B. setPosition() C. location() D. setLocation()

(11) 窗口中 Y 轴位置对应于_____。

- A. 水平位置 B. 垂直位置 C. 字体尺寸 D. 操作顺序

(12) 在屏幕左上角显示 100×100 像素, 它的位置是_____。

- A. 0,0 B. 0,100 C. 100,0 D. 100,100

(13) 在屏幕右上角显示 100×100 像素, 它的位置是_____。

- A. 0,0 B. 0,100 C. 100,0 D. 100,100

(14) 在 200×200 像素的一个窗口中位置 10,190 最接近_____。

- A. 左上 B. 右上 C. 左下 D. 右下

(15) 可使用 setEnabled()方法让一个组件_____。

- A. 起作用 B. 失效 C. 以上两者兼可 D. 以上两者兼不可

(16) 下面哪条语句使名叫 someComponent 的组件失效?

- A. someComponent.setDisabled()
C. someComponent.setDisabled(false)
- B. someComponent.setDisabled(true)
D. someComponent.setDisabled(true)

8. 创建一个名叫 DoubleInteger 的 Applet, 它允许用户输入一个整型数。当用户单击按钮时, 该整数乘 2, 且显示结果。

9. 创建一个名叫 SumIntegers 的 Applet, 它允许用户将两个整型数分别输入各自的文本框中。当用户单击按钮时, 显示两整数的总和。

10. 创建一个名叫 DivideTwo 的 Applet, 它允许用户在两个文本框中各自输入一整型数。单击按钮则第 2 个整数除第 1 个整数, 并将结果显示出来。

11. 修改 10 题所创建的 DivideTwo Applet, 以便用户如果将第 2 个数输入为 0, 按按钮进行除法运算前, Applet 显示“不允许 0 为除数!”的信息。

12. 什么是 JDBC? 它有什么作用?

13. 编出一个使用 JDBC 访问数据库的 Applet 小应用程序, 该小应用程序又由 HTML 调用。

第 4 章 Web 层编程技术

本章主要介绍了 Java EE 的 Web 层编程的 JSP (JavaServer Pages) 技术、Java Servlet 技术、JSTL (JSP Standard Tag Library) 标准标签库技术以及 JSF (JavaServer Faces) 等技术, 其既可独立进行 Java EE Web 网站编程, 又为后面章节的学习提供了 JavaEE Web 层编程技术基础。

4.1 JSP 技术

4.1.1 JSP 简介

JSP 是由 Sun 微系统公司于 1999 年 6 月推出的一项技术, 类似于网景公司的服务器端 Java 脚本语言 Server-side JavaScript (SSJS) 和微软的 Active Server Pages (ASP)。JSP 比 SSJS 和 ASP 具有更好的可扩展性, 并且它不专属于任何一家厂商或某一特定的 Web 服务器。尽管 JSP 规范是由 Sun 公司制定的, 但任何厂商都可以在自己的系统上实现 JSP。利用这一技术可以建立先进、安全和跨平台的动态网站。

1. 什么是 JSP

在 Sun 正式发布 JSP 之后, 这种新的 Web 应用开发技术很快引起了人们的关注。JSP 为创建高度动态的 Web 应用提供了一个独特的开发环境。按照 Sun 的说法, JSP 能够适应市场上包括 Apache WebServer、IIS 在内的 85% 的服务器产品。JSP 提供在 HTML 代码中混合某种程序代码、由语言引擎解释执行程序代码的能力。同时, JSP 是面向 Web 服务器的技术, 客户端浏览器不需要任何附加的软件支持。

说到什么是 JSP, 先来看下面这段代码。

【例 4-1】JSP 程序示例。

```
<%!  
String hello(){  
return "您好,朋友,欢迎进入 JSP 世界,^_^";  
}  
%>  
  
<html>  
<head>  
<title>JSP 例程-在 JSP 中定义函数</title>  
</head>  
<body>  
<%=hello()%>  
</body>  
</html>
```



图 4-1 JSP 示例程序

只要把这段程序作为 JSP 文件放在 Web 服务器能找到它的位置, 就可以在客户浏览器中看到图 4-1。

分析这段代码, 可以看出来, 它和 ASP 文件有很多相似之处, 不过它是以 .jsp 为扩展名的文本文件, 这个文本文件可以包括文本、HTML 标记、JavaScript 及一些更复杂的标记或脚本, 由此组成了一个具有某种特定功能的 JSP 页。

2. JSP的优点

JSP 技术在多个方面加速了动态 Web 页面的开发, 它具有很多优点。

(1) 它可以将内容的生成和显示进行分离。使用 JSP 技术, Web 页面开发人员可以使用 HTML 或者 XML 标识来设计和格式化最终页面; 使用 JSP 标识或者小脚本来生成页面上的动态内容。生成内容的逻辑被封装在标识和 JavaBeans 组件中, 并且捆绑在小脚本中, 所有的脚本在服务器端运行。如果核心逻辑被封装在标识和 Beans 中, 那么其他人, 如 Web 管理人员和页面设计者, 能够编辑和使用 JSP 页面, 而不影响内容的生成。在服务器端, JSP 引擎解释 JSP 标识和小脚本, 生成所请求的内容 (例如, 通过访问 JavaBeans 组件, 使用 JDBC TM 技术访问数据库或者包含文件), 并且将结果以 HTML (或者 XML) 页面的形式发送回浏览器。这有助于作者保护自己的代码, 而又保证任何基于 HTML 的 Web 浏览器的完全可用性。

(2) 强调可重用的组件。绝大多数 JSP 页面依赖于可重用的, 跨平台的组件 (JavaBeans 或者 Enterprise JavaBeans TM 组件) 来执行应用程序所要求的更为复杂的处理。开发人员能够共享和交换执行普通操作的组件, 或者使得这些组件为更多的使用者或者客户团体所使用。基于组件的方法加速了总体开发过程, 并且使得各种组织在他们现有的技能和优化结果的开发努力中得到平衡。

(3) 采用标识简化页面开发。Web 页面开发人员不会都是熟悉脚本语言的编程人员。JavaServer Page 技术封装了许多功能, 这些功能是在易用的、与 JSP 相关的 XML 标识中进行动态内容生成所需要的。标准的 JSP 标识能够访问和实例化 JavaBeans 组件, 设置或者检索组件属性, 下载 Applet, 以及执行用其他方法更难于编码和耗时的功能。

3. JSP的工作原理 (JSP与Servlet的关系)

JSP 技术实际上是通过引擎 JSP 把 JSP 标签、JSP 页中的 Java 代码甚至连同静态 HTML 内容都转换为大段的 Java 代码。这些代码块被 JSP 引擎组织到用户看不到的 Java Servlet 中去, 然后 Servlet 自动把它们编译成 Java 字节码。这样, 当网站的访问者请求一个 JSP 页时, 在他不知道的情况下, 一个已经生成的、预编译过的 Servlet 实际上将完成所有的工作, 非常隐蔽而又高效。因为 Servlet 是编译过的, 所以网页中的 JSP 代码不需要在每次请求该页时被解释一遍。JSP 引擎只需在 Servlet 代码最后被修改后编译一次, 然后这个编译过的 Servlet 就可以被执行了。由于是 JSP 引擎自动生成并编译 Servlet, 不用程序员动手编译代码, 所以 JSP 能提供高效的性能和快速开发所需的灵活性。如图 4-2 所示为 JSP 的工作原理。

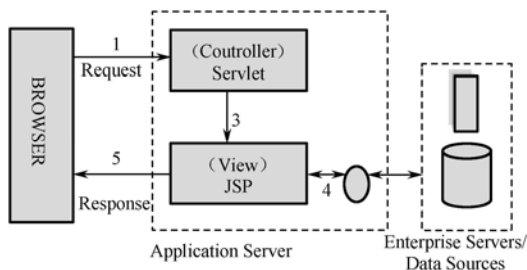


图 4-2 JSP 的工作原理

作为 Java 平台的一部分，JSP 拥有 Java 编程语言“一次编写，各处运行”的特点。随着越来越多的供应商将 JSP 支持添加到他们的产品中，可以使用自己所选择的服务器和工具，更改工具或服务器并不影响当前的应用。

JSP 的强大是因为其后面有强大的 Java 技术支持，包括 JavaBeans 和 Java EE 技术在内的 Java 技术是 JSP 强大生命力所在。

4.1.2 JSP的语法

JSP 与用户的交互过程是基于“请求—响应”模式的。例如，用户如果希望服务器完成一个带有参数的请求，就必须以表单（form）的方式进行提交。表单是 HTML 中的一个基本元素，在实际应用中十分常见，例如，登录网站之前要在表单中填写用户名和密码，进行查询时要填写查询关键词等。在表单中，用户能够输入一些信息，当用户将一个表单填写完毕后可以将表单提交，服务器收到提交上来的请求后，会对表单中的内容进行分析，完成相应的处理后再返回用户希望得到的信息。要完成这些任务，就要合法地编写 JSP 代码，本节介绍 JSP 的基本语法格式。

1. JSP语法概要

JSP 文件（扩展名为.jsp）可以包含指令（或称为指示语句）、变量和方法、直接插入的 Java 代码（scriptlet）、访问 JavaBeans、变量数据的 Java 表达式的组合等。

在解释 JSP 语法前，创建一个 JSP 网页，显示当前的日期和时间，并回显四次问候语。

【例 4-2】 创建一个 JSP 网页。

（文件名为：sample.jsp）

```

<%@ page import="java.util.Date" %>
<%@ page language="java" %>
<html>
<head>
<!-- comment for server side only -->.
    <title>说明语法的示例页</title>
</head>
<body>
    <H3>Today is:
        <%= new java.util.Date() %>
    </H3>
    <% for (int i=1; i<=4; i++) { %>
    <H<%=i%>>Hello</H<%=i%>>

```



```
<% } %>
```

```
</body>
```

```
</html>
```

例 4-2 中看到的语句行：`<%@ page language="java" %>`就是一个 JSP 指令，`<% for... %>` 则是一段 Java 程序代码段。

2. 指令

所谓 JSP 指令是为 JSP 引擎而设计的。它们并不直接产生任何可见输出，而只是告诉引擎如何处理其余 JSP 页面，为页面提供全局信息，如导入语句、错误处理页面或该页面是否为会话的一部分等。常用到的有 Page 指令和 Include 指令。

(1) Page 指令：用 Page 指令来定义 JSP 文件中的全局属性。

从例 4-2 中可以看到其语法格式为：

```
<%@ page att = "val" %> 或 <jsp: directive.page att = "val" %>
```

格式中的 att 代表指令名，并且在格式`<%@ page att = "val" %>`中指令字 page 是可以省略的。常用指令名有如下几种。

① language：指明文件中所用的脚本语言。对于 Java 程序设计语言来说，Java 为有效值和默认值。该命令作用于整个文件，而且当多次使用该指令时，只有第一次使用是有效的。示例如下：

```
<%@ language="java" %>
```

② method：指明由嵌入的 Java 代码(scriptlet)生成的方法的名称。生成的代码会成为指定方法名的主体。默认的方法是 service()。当多次使用该指令时，只有第一次使用是有效的。示例如下：

```
<%@ method="doPost" %>
```

③ import：指明 Servlet 导入的 Java 语言软件包名和类名列表，该列表是用逗号分隔的。在 JSP 文件中，可以多次使用该指令来导入不同的软件包。示例如下：

```
<%@ import="java.io.*, java.Util.Hashtable"%>
```

④ content type：指明生成响应的 MIME 类型，默认值为 text/html。当多次使用该指令时，只有第一次使用是有效的，该指令还可用以指定对页面进行编码的字符集。示例如下：

```
<%@ content_type="text/html; charset=gb2312" %>
```

⑤ implements：指明用于生成 Servlet 实现的 Java 语言接口列表，该列表是用逗号分隔的，可以在一个 JSP 文件中多次使用该命令，以实现不同的接口。示例如下：

```
<%@ implements="javax.Servlet.http.HttpSessionContext"%>
```

⑥ extends：指明 Servlet 扩展的 Java 语言类的名称。该类必须是有效的，且不能是一个 Servlet 类。该指令作用于整个 JSP 文件，而且当多次使用该指令时，只有第一次使用是有效的。示例如下：

```
<%@ extends="javax.Servlet.http.HttpServlet"%>
```

无论把`<% @ page ... %>`指令放在 JSP 的文件的哪个地方，它的作用范围都是整个 JSP 页面。不过，为了 JSP 程序的可读性，以及好的编程习惯，最好还是把它放在 JSP 文件的顶部。

(2) Include 指令：在 JSP 中包含一个静态的文件，同时解析这个文件中的 JSP 语句。

使用格式为：

```
<%@ include file="relativeURL" %>
```

其中 relativeURL 代表被包含的文件的路径。

【例 4-3】Include 指令的使用。

```
<html>
<head>
<title>An Include Test</title>
</head>
<body bgcolor="white">
<font color="blue"> The current date and time are
<%@ include file="date.jsp" %>
</font>
</body>
</html>
```

上述 HTML 文件中所包含的 date.jsp 文件代码如下 (date.jsp)：

```
<%@ page import="java.util.*" %>
<%= (new java.util.Date() ).toLocaleString() %>
```

<%@ include file="relativeURL" %>指令将会在 JSP 编译时插入一个包含文本或代码的文件，当使用<%@ include %>指令时，这个包含的过程是静态的。静态的包含就是指这个被包含的文件将会被插入到 JSP 文件中去，这个包含的文件可以是 JSP 文件、HTML 文件、文本文件。如果包含的是 JSP 文件，这个包含的 JSP 的文件中代码将会被执行。而如果仅仅只是用 include 来包含一个静态文件，那么这个包含的文件所执行的结果将会插入到 JSP 文件中放<%@ include file="relativeURL" %>的地方。

一旦包含文件被执行，那么主 JSP 文件的过程将会被恢复，继续执行下一行。但必须注意在包含文件中不能使用<html>，</html>，<body>，</body>标记，因为这将会影响在原 JSP 文件中同样的标记，这样做有时会导致错误。

3. 声明

用来在 JSP 程序中声明合法的变量和方法。语法格式为：

```
<%! Declaration; [ declaration; ]; ...; %>
```

使用的方法如下例：

```
<%! int i = 0; %>
<%! int a, b, c; %>
<%! Circle a = new Circle(2.0); %>
```

用这种方法可以声明将要在 JSP 程序中用到的变量和方法，可以一次性声明多个变量和方法，但每个变量和方法要以";"来分割，并且最后以";"结尾。

也可以直接使用在<% @ page... %>中被包含进来的已经声明的变量和方法，不需要对它们重新进行声明。

一个声明仅在一个页面中有效。如果想每个页面都用到一些声明，最好把它们写成一个单独的文件，然后用<%@ include... %>或<jsp:include... >元素将其包含进来。

4. 插入代码段

JSP 代码片段或脚本片段 (scriptlet) 是嵌在 “<% 代码 %>” 标记中的。语法格式为:

<% 代码段 %> 或 <jsp:scriptlet>代码</jsp:scriptlet>

在 JSP 代码段中插入用于服务的代码。这里的代码与平常的 Java 代码一样, 每条语句必须以分号结束。

这种 Java 代码在 Web 服务器响应请求时就会运行。在脚本片段周围可能是 HTML 或 XML 语句, 在这些地方, 代码片段可以创建条件执行代码, 或要用到另外一段代码的代码。例如, 以下的代码组合使用表达式和代码片段, 显示 H1、H2、H3 和 H4 标记中的字符串 “Hello”。代码片段并不局限于一行源代码:

```
<% for (int i=1; i<=4; i++)
{ %>
<H<%=i%>>Hello</H<%=i%>>
<% } %>
```

5. 注释

JSP 语法中还有一个重要的元素就是注释。注释的目的是对 JSP 页面功能进行说明, 主要面向程序员和维护人员, 所以尽管可以在文件中加入 HTML 注释, 但用户在查看页面源代码时会看到这些注释。如果不想让用户看到它, 就应该将其嵌入 <%... 注释 ...%> 标记中, 这个标记是在 JSP 技术中特有的, 另外还可以使用普通的 Java 所用的两种注释方法, 即:

```
<%...
// 注释
....%>
<%...
/*... 注释 ...*/
...%>
```

在 JSP 中, 随着功能的增强, 可能还会用到一些指令, 如 forward 指令, 它可以重定向到一个 HTML 文件、JSP 文件、或者是一个程序段; 用 getProperty 指令获取 Bean 的属性值或用 setProperty 设置 Bean 中的属性值; 用 plugin 指令在浏览器中播放或显示一个 applet 或 Bean 等对象。这些内容将在以后的相关章节代码中看到。

4.1.3 JSP的内建对象

JSP 共有以下 9 种基本内建对象。

- request: 用户端请求, 此请求包含来自 GET/POST 请求的参数;
- response: 网页传回对用户端的回应;
- pageContext: 网页的属性在此管理;
- session: 与请求有关的会话期;
- application: Servlet 正在执行的内容;
- out: javax.jsp.JspWriter 的一个实例, 提供了向浏览器回送输出结果的方法;
- Config: 用来表示 Servlet 的构架部件;
- page JSP: 网页本身;

- **Cookie:** 用来将少量信息保存到浏览器中。

使用内建对象可以存取执行 JSP 代码的 Servlet。采用内建对象的最大优点是可以简化网页代码的编写。例如，可以直接输出 response:

```
<% out.println("Hello"); %>
```

也可以不必直接传送参数到 JavaBean，而是按照请求来取得参数的值:

```
<% String name=request.getParameter("name");  
out.println(name); %>
```

以下以 session 对象及 application 为例来说明它们的应用。

会话状态维持是很多 Web 应用开发者所关心的问题。有多种方法可以用来解决这个问题，例如，使用 Cookies、隐藏的表单输入域，或直接将状态信息附加到 URL 中。session 对象为 JSP 的应用开发较好地解决了会话状态维持的问题。当客户首次访问服务器上的一个 JSP 页面时，JSP 引擎产生一个 session 对象，这个 session 对象调用相应的方法可以存储客户在访问各个页面期间提交的各种信息，并且这个 session 对象被分配了一个 String 类型的 ID 号，JSP 引擎同时将这个 ID 号发送到客户端，存放在客户的 Cookie 中。该次会话的所有页面都将使用该 ID。JSP 中的 session 对象对于那些希望通过多个页面完成一个事务的应用是非常有用的。

为说明 session 对象的具体应用，接下来用三个页面模拟一个多页面的 Web 应用。

第一个页面 (q1.html) 仅包含一个要求输入用户名字的 HTML 表单，代码如下:

```
<HTML>  
<BODY>  
<FORM METHOD=POST ACTION="q2.jsp">  
请输入您的姓名:  
<INPUT TYPE=TEXT NAME="thename">  
<INPUT TYPE=SUBMIT VALUE="SUBMIT">  
</FORM>  
</BODY>  
</HTML>
```

第二个页面是一个 JSP 页面 (q2.jsp)，它通过 request 对象提取 q1.html 表单中的 thename 值，将它存储为 name 变量，然后将这个 name 值保存到 session 对象中。session 对象是一个名字/值对的集合，在这里，名字/值对中的名字为“thename”，值即为 name 变量的值。由于 session 对象在会话期间是一直有效的，因此这里保存的变量对后继的页面也有效。q2.jsp 的另外一个任务是询问第二个问题。下面是它的代码:

```
< HTML>  
< BODY>  
< %@ page language="java" %>  
< %! String name=""; %>  
< %  
name = request.getParameter("thename");  
session.putValue("thename", name);  
%>  
您的姓名是: < %= name %>  
< p>
```

```

< FORM METHOD="POST" ACTION="q3.jsp">
您喜欢什么活动 ?
< INPUT TYPE=TEXT NAME="action">
< P>
< INPUT TYPE=SUBMIT VALUE="SUBMIT">
< /FORM>
< /BODY>
< /HTML>

```

第三个页面也是一个 JSP 页面 (q3.jsp)，主要任务是显示问答结果。它从 session 对象提取 thename 的值并显示它，以此证明虽然该值在第一个页面输入，但通过 session 对象得以保留。q3.jsp 的另外一个任务是提取在第二个页面中的用户输入并显示它：

```

< HTML>
< BODY>
< %@ page language="java" %>
< %! String act=""; %>
< %
act = request.getParameter("action");
String name = (String) session.getValue("thename");
您的姓名是: < %= name %>
< P>
您喜欢的活动是: < %= act %>
</BODY>
</HTML>

```

有时候服务器需要维护全局的一些数据，例如，服务器端希望做一个计数器，记录每一个页面的被访问次数，这时可以使用内建对象 application。当服务器启动时，application 对象就会被创建，它的生存期将一直持续到服务器关闭。因此，在服务器工作期间，application 对象能够起到保存信息的作用，而且该对象只有一个实例，能够保证信息同步。application 的两个主要的方法是 setAttribute 和 getAttribute，分别用来存储/读取一个属性。下面的程序显示如何使用 application 对象进行全局信息的维护。

【例 4-4】显示如何使用 application 对象进行全局信息的维护。

```

<%
Integer number = (Integer)application.getAttribute("num");
if(number == null){
number = new Integer(1);
}else{
number = new Integer(number.intValue() + 1);
}
application.setAttribute("num",number);
% >
<html>
<head>
<title> application </title>
</head>
<body>

```

```
<h1> application</h1>
该页面自服务器启动以来已被访问过<% = number%>次!
</body>
</html>
```

这个 JSP 页面首先检查 application 对象中是否已经设置了 num 属性，如果没有，那么就将其初始化，即“如果 num 已经存在，那么将 num 的值加 1”，这样就实现了一个简单的计数器。多次访问该页面的结果如图 4-3 所示。



图 4-3 有简单计数功能的页面

此后对该页面每访问一次，访问次数会加 1。

4.1.4 JSP的表单及Cookie应用

1. 表单

表单是 HTML 的元素，在 HTML 或者 JSP 中都可以很容易生成一个表单。在一个 HTML 或者 JSP 页面中可以有很多表单，但是在提交表单的时候一般只提交一个表单。此外，表单不能够嵌套使用。表单在 HTML 中是以<form>...</form>标记的。

【例 4-5】一个简单的表单。

```
<html>
  <head>
    <title>form</title>
  </head>
  <body>
    <h1>This is a form</h1>
    <form name = "myform" action = "getform.jsp" method = "post">
      <input name = "username">
      <input type = submit value = "提交">
    </form>
  </body>
</html>
```

在 form 开始标签后面的“name”是该表单的名字，“action”表示提交后应该采取什么动作，如果在后面的代码是 URL，则表示将该表单提交给这个 URL，“method”表示该表单以什么方式提交。

表单的提交通常有 Get 和 Post 两种方法。这两种方法区别在于：使用 Get 方法提交的数据会在 URL 地址栏中显示出来，而使用 Post 方法提交的数据是不会显示出来的。Post 方法没有数据类型和数据量的限制，而 Get 方法只能提交文本类型的数据。其中数据量的限制

也就是 URL 长度的限制，一般为 2048 字节。

由于 Get 方法提交的数据会出现在 URL 地址栏中，而一般的浏览器都有自动记录 URL 地址的功能，所以使用 Get 方法提交数据存在不安全性，而与安全相关的数据必须使用 Post 方法来提交。

在页面中，表单本身是不可见的，它只不过是作为一个框架将其内部的控件包含起来，形成一个整体，在提交的时候，这个表单内部的所有控件中的内容都会被一同提交。而表单内部的控件在页面中是可见的，是需要浏览者对其实施操作的。

现在要用 JSP 技术来处理提交上来的表单。下面的例子是用来在浏览器中显示表单提交的所有参数。

【例 4-6】在浏览器中显示表单提交的所有参数，文件名保存为 getform.jsp。

```
1 <html>
2 <head>
3 <title>show form</title>
4 </head>
5 <body>
6 <h1>This is submitted by a form</h1>
7 <%
8 //得到所有的参数名称
9 java.util.Enumeration e = request.getParameterNames();
10 //对所有参数进行循环
11 while(e.hasMoreElements( )){
12 //得到参数名
13 String name = (string)e.nextElement( );
14 //得到这个参数的所有值
15 String[] value = request.getParameterValues(name);
16 //打印参数名
17 out.print(name + ":");
18 //对一个参数的所有的值进行循环
19 for(int i = 0;i<value.length;i++){
20 //打印一个参数值
21 out.print(value[ i ]);
22 if(i != value.length-1)
23 out.print(",");
24 }
25 out.print("<br>");
26 }
27 %>
28 </body>
29 </html>
```

第 15 行之所以要用一个字符串数组来接收一个参数的值是因为一个请求中参数可以有很多个值。使用 request.getParameterValues(Stringname)方法就可以得到名为“name”的参数的所有值，而使用 request.getParameter(Stringname)仅能得到该参数的第一个值。

表单内部由很多控件组成，表单的控件有很多种，每种控件在页面上的表现形式各不

相同，作用也有所区别。

2. Cookie

Cookie 是一种保存持续状态信息和其他信息的一种方式，它表示由 Web 浏览器存储起来的一小组数据，通过这些数据可以在 Web 页内共享信息。一个典型的 Cookie 中存储着与某个站点相关的用户信息，以便将来连接到该站点时使用。

Cookie 保存的路径为：

Windows 98，%Windir%\Temporary Internet Files\。

Windows 2000 以上，%USERPROFILE%\Local Settings\Temporary Internet Files\。

Cookie 由一个或多个 NAME=VALUE 对组成，并包括几个特殊字段：expires、domain、path 和 secure。

expires 字段：表示 Cookie 的过期日期。如果不指定此字段，则 Cookie 在当前浏览器会话结束时过期。为了使 Cookie 的持续时间变长，应将 expires 字段设置为合法的时间值。如果指定了 expires 字段，则浏览器会话结束后，Cookie 被写到文本文件中；否则浏览器会话结束后，Cookie 自动销毁。

domain 字段：表示 Cookie 的有效域。例如，如果指定 domain=www.MySite.com，则 Cookie 被该域的所有网页共享。

path 字段：确定比 domain 更进一步的有效范围，取值为文件夹名。例如，如果同时指定 domain=www.Mysite.com 和 path=/examples，则表示 Cookie 在 www.Mysite.com/examples 范围内有效。

secure 字段：如果指定此字段（不需要取值），则 Cookie 只能在安全通信通道（HTTPS）上传递。

使用 Cookie 涉及向客户端输出 Cookie 和在服务端读出 Cookie 的过程。

（1）向客户端输出 Cookie。

① 创建 Cookie 对象：

```
Cookie mycookie = new Cookie("username", "albatross");
```

② 设置过期时间：

```
mycookie.setMaxAge(365*24*60*60);
```

③ 设置域：

```
mycookie.setDomain("mysite.com");
```

④ 设置路径：

```
mycookie.setPath("/test");
```

⑤ 向客户端写 Cookie：

```
response.addCookie(mycookie);
```

（2）在服务端读出 Cookie。

① 读出客户端的 Cookie，取得所有可读 Cookie：

```
Cookie[] cookies = request.getCookies();
```

② 取出指定的 Cookie，取得 Cookie 名字：

```
cookies[i].getName();
```

③ 取得 Cookie 的值：

```
cookies[i].getValue();
```


【例 4-7】一个 Cookie 的应用实例。

```
// 这是一段向客户端写入 Cookie 的值的代码
<%
    Cookie mycookie = new Cookie("username", "albatross");
    mycookie.setMaxAge(365*24*60*60);
    response.addCookie(mycookie);
    out.print("Cookie has been setted.");
%>
//这一段代码的作用是读出客户端 Cookie
<@ page contentType = "text/html; charset = gb2312" %>
<%
    Cookie[] cookies = request.getCookies();
    out.print("<h1>读取客户端 Cookie 信息\n");
    for (int i = 0; i < cookies.length; i++)
    {
        out.print("<p>" + cookies[i].getName() + "\t"
            + cookies[i].getValue());
    }
    if (cookies.length <= 0)
        out.print("No cookies!");%>
```

4.1.5 JSP与JavaBean

当然可以把大段的代码放在脚本片段（scriptlet）内，但是绝大多数的 Java 代码属于可重复使用的 JavaBean 的组件。JavaBean 类似于 ActiveX 控件，它们都能提供常用功能并且可以重复使用。

1. 什么是JavaBean

JavaBean 是一个特殊的类，这个类必须符合 JavaBean 规范。它是一个对象，一段独立的代码，定义了一个现实世界的事物或概念，表达了一个概念性的任务，可在不同的应用里使用。传统的 JavaBean 是为了能够在一个可视化的集成开发环境中可视化、模块化地利用组件技术开发应用程序而设计的。而在 JSP 技术中 JavaBean 组件则有可视化的和非可视化两种应用领域。所谓可视化的 JavaBean 组件是指在运行中能够显示出来，如文本框、按钮等；而非可视化的 JavaBean 组件通常用来处理程序中的一些复杂事务，一般不会有可视化的输出。

JavaBean 的主要特征包括属性、方法和事件。

由于 JSP 常用 JavaBean 来封装事务逻辑、数据库操作等，可以很好地实现逻辑和前台程序的分离，使得系统更加健壮和灵活，所以在服务器端应用方面表现出了越来越强的生命力。

编写 Bean 的最好方法是遵循 Sun Microsystems 的 JavaBean 规范。JavaBean 的优势在于 Java 带来的可移植性。

JavaBean 其实是一个放置在 JSP 服务器后端的一个类，它封装了一些私有的数据和和方法，这些数据在 JSP 页面中可以通过 set 和 get 方法来存取。它的工作原理是：首先在 JSP 页面中生成一个 JavaBean，再调用某些 JavaBean 内部的方法来对数据进行处理，最后使用 get 方法得到最终结果在页面上显示出来。它的工作原理如图 4-4 所示。

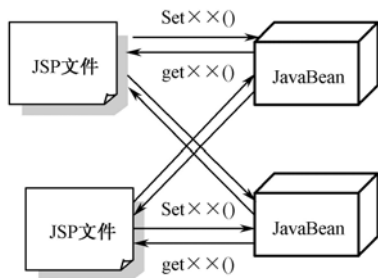


图 4-4 JavaBean 的工作原理

2. 编写JavaBean源文件

为了说明 JavaBean 的应用方法，下面先来看一个简单实例。

【例 4-8】 编写一个简单的 JavaBean (JavaBean 源文件部分)。

```

package bean;
public class sampleBean
{
    private String sample="Start value";
    public String getSample()
    {
        return sample;
    }
    public void setSample(String newValue)
    {
        if(newValue!=null)
        {
            sample=newValue ;
        }
    }
}

```

把编写好的 JavaBean 按类名 sampleBean 保存好。要注意使用 JavaBean 时，其存取源文件名必须和 JavaBean 的类名一致。

从例子中看到，编写一个 JavaBean 就和编写一段普通的 Java 类一样，没有太大区别。但要注意的，如果类的成员变量的名字是 xxx，那么为了更改或获取成员变量的值，即更改或获取属性，在类中只能使用 getxxx() 用于获取属性 xxx，用 setxxx() 方法来修改属性 xxx。

接下来进入 DOS 命令行状态，编译写好的 JavaBean 的代码，命令行方式为：

```
javac sampleBean.java -d .
```

编译成功后，在当前目录下会生成一个名称为 bean 的下级目录，在这个目录中包含了编译好的 Java 字节码文件 sampleBean.class。把这个目录及目录中的文件一起复制到 Tomcat 安装目录\webapps\自建目录\WEB-INF\classes 下（或放在 Tomcat 的安装目录\shared\classes 下），然后重新启动 Tomcat 服务器。

【例 4-9】 访问 JavaBean 的 JSP 源代码部分。

```

<%@ page contentType="text/html;charset=GBK"%>
<html>
<head>

```

```

<title>Servlet use JSP</title>
</head>
<jsp:useBean id="myBean"
    scope="application" class="bean.sampleBean"/>
<body bgcolor="gray">
    <h2>JSP use javaBeab:</h2>
    <hr>
    jsp:before
    <jsp:getProperty name="myBean" property="sample"/>
    <p>
    <jsp:setProperty name="myBean"
        property="sample" value="dhdhfhdfhd"/>
    jsp:after
    <jsp:getProperty name="myBean1" property="sample"/>
</body>
</html>

```

总结以上实例，可以看到：很多时候可通过使用 **JavaBean** 来使网站达到动态化效果。首先要将 **JavaBean** 创建出来，创建的方法是写一个 **JavaBean** 的 **Java** 源代码，然后把它编译成相应的 **class** 字节码存放在正确的位置。如果以 **Tomcat** 作为 **Web** 服务器，要使用 **JavaBean** 必须手工编译，本例中这个 **Java** 类命名为 **sampleBean.java**，编译好后产生的 **sampleBean.class** 就是一个可用的 **JavaBean**，将其放在 **Tomcat** 的安装目录\shared\classes 下，就可以编写 **JSP** 来调用它了。

接着编写 **JSP** 代码，在 **JSP** 文件中，需要做的事就是告诉 **JSP** 页面，程序将要用到一个“**Bean**”，使用的格式如下：

```

<jsp:useBean id="给 bean 起的名字"
    class="创建 bean 的类" scope="bean 的有效范围">
</jsp:useBean>

```

或

```

<jsp:useBean id="给 bean 起的名字"
    class="创建 bean 的类" scope="bean 的有效范围"/>

```

例如：

```

<jsp:useBean id="myBean"
    scope="application" class="bean.sampleBean"/>

```

<jsp:useBean> 标记要求用“**id**”属性来识别 **JavaBean**。这里提供一个名字来让 **JSP** 页面其余部分识别 **Bean**。除了“**id**”属性，还必须告诉页面从何处查找 **JavaBean**，或者它的 **Java** 类别名是什么。这种类别属性提供确认 **Bean** 的功能，其他一些方法也可以做到这一点。最后一个必需的元素是“**scope**”属性。有了“**scope**”属性的帮助，就能告诉 **Bean** 为单一页面（默认情况）[**scope**="page"]、为[**scope**="request"]请求、为会话[**scope**="session"]、或者为整个应用程序[**scope**="application"]保留信息。

当声明了一个 **JavaBean** 时，就可以访问它的属性来定制它。要获得属性值，可以用以下两种格式中的任意一种：

```

<jsp:getProperty name="beans 的名字" property="beans 的属性"/>

```

或

```
<jsp:getProperty name="beans 的名字"
    property="beans 的属性">
```

```
</jsp:getProperty>
```

同样，要改变 JavaBean 属性，必须使用如下格式：

```
<jsp:setProperty name="beans 的值"    property="beans 的属性名"
    value="<%=expression%>" />
```

【例 4-10】定制 JavaBean 属性值。

```
package test ;

public class TestBean{
    private String stringValue;
    private int num;
    //初始化
    public TestBean(){
        stringValue="This is the initial value.";
        num=0;
    }
    //设置 value
    public void setValue(String avalue){
        stringValue=avalue;
    }
    //得到 value
    public String getValue() {
        return stringValue;
    }
    //设置 number
    public void setNumber(int number){
        num=number;
    }
    //得到 number
    public int getNumber() {
        return num;
    }
}
```

定制 TestBean 的 JSP 代码部分

```
<html>
<head>
<title>page</title>
</head>
<body>
<h1>This page uses a javabean.</h1>
<jsp:useBean id="pagebean" class="test.TestBean" scope="page" />
<%=pagebean.getValue()%>
</body>
</html>
```

4.2 Java Servlet技术

4.2.1 Servlet概述

1. 什么是Servlet

Servlet 是使用 Java Servlet 应用程序设计接口(API)相关类和方法的 Java 程序，它运行于 Web 的服务端。除了 JavaServletAPI，它还可以使用各种扩展和添加到 API 的 Java 类软件包。Servlet 在启用 Java 的 Web 服务器上或应用服务器上运行并扩展了该服务器的能力。Java Servlet 对于 Web 服务器就好像 Java Applet 对于 Web 浏览器一样。Servlet 装入 Web 服务器并在 Web 服务器内执行，而 Applet 则装入 Web 浏览器并在 Web 浏览器内执行。JavaServletAPI 定义了 Servlet 和 Java 使用的服务器之间的一个标准接口，这使得 Servlet 具有跨服务器平台的特性。

前面介绍的 JSP 程序在执行时，首先被支持的 JSP 容器如 Tomcat、Resin 等进行预编译，生成一个标准的 Java 程序 (Servlet)，然后 JSP 容器会把这个 Java 程序执行的结果以 HTML 的格式返回给浏览器进行显示。JSP 与 Servlet 既有关联，又有区别。JSP 更注重页面的表现，而 Servlet 更注重业务逻辑的实现。JSP 只能处理客户的请求，而 Servlet 还可以处理客户端的应用程序请求。

Servlet 通过创建一个框架来扩展服务器的能力，以提供在 Web 上进行请求和响应的服务。当客户机发送请求至服务器时，服务器可以将请求信息发送给 Servlet，并让其建立起服务器返回给客户机的响应。当启动 Web 服务器或客户机第一次请求服务时，可以自动装入 Servlet。装入后，Servlet 继续运行直到其他客户机发出请求。Servlet 的功能涉及范围很广。例如，它可完成如下功能：

- (1) 创建并返回一个包含基于客户请求性质的动态内容的完整 HTML 页面。
- (2) 创建可嵌入到现有 HTML 页面中的一部分 HTML 页面 (HTML 片段)。
- (3) 与其他服务器资源 (包括数据库和基于 Java 的应用程序) 进行通信。
- (4) 处理与多个客户机的连接，接收多个客户机的输入，并将结果广播到多个客户机上，例如，Servlet 可以作为许多参与者的游戏服务器。
- (5) 当允许在单连接方式下传送数据时，可以在浏览器上打开服务器至 Applet 的新连接，并将该连接保持在打开状态。当允许客户机和服务器简单、高效地执行会话时，Applet 也可以启动客户浏览器和服务器之间的连接，并且可以通过定制协议或标准 (如 IIOP) 进行通信。
- (6) 对特殊的处理采用 MIME (MIME 表示多用途 Internet 邮件扩充协议) 类型过滤数据。
- (7) 将定制的处理提供给所有服务器的标准例行程序。例如，Servlet 可以修改如何认证用户。

为了进一步认识 Servlet，先来看一个最简单的 Servlet 的例子。

【例 4-11】简单的 Servlet 例子。

```
Hello.java
import java.io.*;
```

```

import javax.servlet.*;
import javax.servlet.http.*;
public class Hello extends HttpServlet
{
    public void init(ServletConfig config)
        throws ServletException
    {
        super.init(config);
    }
    public void service(HttpServletRequest request, HttpServletResponse response)
        throws IOException
    {
        PrintWriter out = response.getWriter();
        response.setContentType("text/html; charset=GB2312");
        out.println("<html><body>");
        out.println("Simple servlet");
        out.println("</body></html>");
    }
}

```

2. Servlet的生命周期

Servlet 的生命周期始于将它装入 Web 服务器的内存时，并在终止或重新装入 Servlet 时结束，分为初始化、请求处理和终止三个阶段。

(1) 初始化。在下列时刻装入 Servlet:

- ① 如果已配置自动装入选项，则在启动服务器时自动装入；
- ② 在服务器启动后，客户机首次向 Servlet 发出请求时；
- ③ 重新装入 Servlet 时。装入 Servlet 后，服务器创建一个 Servlet 实例并且调用它的 init()方法，如例 4-11 中的 super.init(config)。

(2) 请求处理。对于到达服务器的客户机请求，服务器创建特定于请求的一个“请求”对象和一个“响应”对象。服务器调用 Servlet 的 service()方法，例如，例 4-11 中的 public void service(HttpServletRequest request, HttpServletResponse response)，该方法用于传递“请求”和“响应”对象。service()方法从“请求”对象获得请求信息，同时处理该请求，并用“响应”对象的方法将响应传回客户机。service()方法也可以调用其他方法来处理请求，如用 doGet()、doPost()等。

(3) 终止。当服务器不再需要 Servlet，或重新装入 Servlet 的新实例时，服务器会调用它的 destroy()方法，来终止当前 Servlet 实例的运行。

3. Java Servlet API

Java Servlet 开发工具 (JSDK)提供了多个软件包，在编写 Servlet 时需要用到这些软件包，其中包括两个用于所有 Servlet 的基本软件包：javax.Servlet 和 javax.Servlet.http。

可以从 SUN 公司的 Web 站点下载 Java Servlet 开发工具。下面主要介绍 javax.Servlet.http 提供的 HttpServlet 应用编程接口。

HttpServlet 使用一个 HTML 表单来发送和接收数据。要创建一个 HttpServlet，扩展 HttpServlet 类，该类是用专门的方法来处理 HTML 表单的 GenericServlet 的一个类。HTML 表单是由<FORM>和</FORM>标记定义的。表单中通常包含输入部分（如文本输入框、复选框、单选按钮和选择列表）和用于提交数据的按钮。当提交信息时，它们还指定服务器应执行哪一个 Servlet（或其他程序）。HttpServlet 类包含 init()、destroy()、service()等方法，其中 init()和 destroy()方法是继承的。

(1) init()方法。在 Servlet 的生命周期中，仅执行一次 init()方法。它是在服务器装入 Servlet 时执行的。可以配置服务器，以便在启动服务器或客户机首次访问 Servlet 时装入 Servlet。而且无论有多少客户机访问 Servlet，都不会重复执行 init()方法。

(2) service()方法。service()方法是 Servlet 的核心。每当一个客户请求一个 HttpServlet 对象时，该对象的 service()方法就要被调用，而且传递给这个方法一个“请求”（ServletRequest）对象和一个“响应”（ServletResponse）对象作为参数。在 HttpServlet 中已存在 service()方法。默认的服务功能是调用与 HTTP 请求的方法相应的 do 功能。例如，如果 HTTP 请求方法为 GET，则默认情况下就调用 doGet()方法。Servlet 应该为 Servlet 支持的 HTTP 方法覆盖 do 功能。因为 HttpServlet.service()方法会检查请求方法是否调用了适当的处理方法，因此不必覆盖 service()方法，只需覆盖相应的 do 方法即可。

① 当一个客户通过 HTML 表单发出一个 HTTP POST 请求时，doPost()方法被调用。与 POST 请求相关的参数作为一个单独的 HTTP 请求从浏览器发送到服务器。当需要修改服务器端的数据时，应该使用 doPost()方法。

② 当一个客户通过 HTML 表单发出一个 HTTP GET 请求或直接请求一个 URL 时，doGet()方法被调用。

与 GET 请求相关的参数添加到 URL 的后面，并与这个请求一起发送。当不需要修改服务器端的数据时，应该使用 doGet()方法。

Servlet 的响应可以是下列两种类型：

- 一个输出流，浏览器根据它的内容类型（如 text/HTML）进行解释。
- 一个 HTTP 错误响应，重定向到另一个 URL、Servlet、JSP。

(3) destroy()方法。destroy()方法仅执行一次，即在服务器停止且卸装 Servlet 时执行该方法。典型情况下，将 Servlet 作为服务器进程的一部分来关闭。默认的 destroy()方法是符合要求的，但也可以覆盖它，典型的示例是管理服务器端资源。例如，如果 Servlet 在运行时累计统计数据，则可以编写一个 destroy()方法，该方法用于在未装入 Servlet 时将统计数据保存在文件中。另一个示例是关闭数据库连接。

当服务器卸装 Servlet 时，将在所有 service()方法调用完成后，或在指定的时间间隔过后调用 destroy()方法。一个 Servlet 在运行 service()方法时可能会产生其他的线程，因此应该确认在调用 destroy()方法时，这些线程已终止或完成。

(4) GetServletConfig()方法。GetServletConfig()方法返回一个 ServletConfig 对象，该对象用来返回初始化参数和 ServletContext。ServletContext 接口提供了有关 Servlet 的环境信息。

(5) GetServletInfo()方法。GetServletInfo()方法是一个可选的方法，它提供有关 Servlet 的信息，如作者、版本、版权。

当服务器调用 Servlet 的 service()、doGet()和 doPost()这三个方法时，均需要“请求”和“响应”对象作为参数。“请求”对象提供有关请求的信息，而“响应”对象提供了一个将响应信息返回给浏览器的通信途径。Javax.Servlet 软件包中的相关类为 ServletResponse 和

ServletRequest，而 javax.servlet.http 软件包中的相关类为 HttpServletRequest 和 HttpServletResponse。Servlet 通过这些对象与服务器通信并最终与客户机通信。Servlet 能通过调用“请求”对象的方法获知客户机环境、服务器环境的信息和所有由客户机提供的信息。Servlet 可以调用“响应”对象的方法发送响应，该响应是准备发回客户机的。

创建一个 HttpServlet，通常涉及下列四个步骤：

- 扩展 HttpServlet 抽象类。
- 重载适当的方法。例如，覆盖（或称为重写）doGet()或 doPost()方法。
- 如果有 HTTP 请求信息的话，获取该信息。用 HttpServletRequest 对象来检索 HTML 表单所提交的数据或 URL 上的查询字符串。“请求”对象含有特定的方法以检索客户机提供的信息，有 getParameterNames()、getParameter()、getParameterValues()等三个方法。
- 生成 HTTP 响应。HttpServletResponse 对象生成响应，并将它返回到发出请示的客户机上。它的方法允许设置“请求”标题和“响应”主体。“响应”对象还含有 getWriter()方法以返回一个 PrintWriter 对象，使用该对象的 print()和 println()方法可以编写 Servlet 响应来返回给客户机。或者，直接使用 out 对象输出有关 HTML 文档内容。

为了进一步理解上面介绍的各种方法，下面再来介绍一个 Servlet 实例。

【例 4-12】一个有关 Servlet 的实例（ServletSample.java）。

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletSample extends HttpServlet{
//第一步:扩展 HttpServlet 抽象类

public void doGet(HttpServletRequest request,HttpServletResponse response)
throws ServletException,IOException{
//第二步:重写 doGet()方法

String myName="" ;
//第三步:获取 HTTP 请求信息

Java.util.Enumeration keys=request.getParameterNames();
while(keys.hasMoreElements());
{
    key=(String)keys.nextElement();
    if (key.equalsIgnoreCase("myName"))
        myName=request.getParameter(key);
}
if (myName=="")
myName="Hello";
//第四步:生成 HTTP 响应

Response.setContentType("text/html");
Response.setHeader("Pragma","No-cache");
Response.setDateHeader("Expires",0);
Response.setHeader("Cache-Control","no-cache");
```



```

Out.println("<head><title>Just a basic Servlet</title></head>");
Out.println("<body>");
Out.println("<h1>Just a basic Servlet</h1>");
Out.println("<p>" +myName+ ",this is a very basic Servlet that writes an
            HTML page.");
Out.println("<p>For instructions on running those samples on your
            WebSphere 应用服务,"+ "open the page;");
out.println("<pre>http://<em>your.server.name</em>/IBMWebAs/samples/index.
            html</pre>");
out.println("where<em>yonr.server.name</em>is the hostname of your Web
            Sphere 应用服务器.");
out.Println("</body></html>");
out.flush();
}
}

```

上述 ServletSample 类扩展 HttpServlet 抽象类，重写 doGet()方法。在重写的 doGet()方法中，获取 HTTP 请求中的一个任意的参数(myName)，该参数可作为调用的 URL 上的查询参数传递到 Servlet。

4. 由URL调用Servlet

要调用 Servlet 或 Web 应用程序，可以使用下列任一种方法：由 URL 调用、在<FORM>标记中调用、在<SERVLET>标记中调用、在 JSP 文件中调用、在 ASP 文件中调用。

用 Servlet 的 URL 从浏览器中调用该 Servlet 的方法有两种：

(1) 指定 Servlet 名称。当用 WebSphere 应用服务器管理器来将一个 Servlet 实例添加(注册)到服务器配置中时，必须指定“Servlet 名称”参数的值。例如，可以指定将 hi 作为 HelloWorldServlet 的 Servlet 名称。要调用这个 Servlet，需打开 http://your.server.name/Servlet/hi，也可以指定 Servlet 和类使用同一名称(HelloWorldServlet)。在这种情况下，将由 http://your.Server.name/Servlet/HelloWorldServlet 来调用 Servlet 的实例。

(2) 在<FORM>标记中指定 Servlet。在<FORM>标记中可以调用 Servlet。HTML 格式使用户能在 Web 页面（即从浏览器)上输入数据，并向 Servlet 提交数据。例如：

```

<form method="GET" action="/Servlet/myServlet">
<ol>
<INPUT TYPE="radio"NAME="broadcast"VALUE="am">AM<BR>
<INPUT TYPE="radio"NAME="broadcast"VALUE="fm">FM<BR>
</ol>
<!--用于放置文本输入区域的标记、按钮和其他的提示符。--!>
</form>

```

Action 特性表明了用于调用 Servlet 的 URL；关于 method 的特性，如果用户输入的信息是通过 get 方法向 Servlet 提交的，则 Servlet 必须优先使用 doGet()方法。反之，如果用户输入的信息是通过 post 方法向 Servlet 提交的，则 Servlet 必须优先使用 doPost()方法。使用 get 方法时，用户提供的信息是用查询字符串表示的 URL 编码。但用户无须对 URL 进行编码，因为这是由表单完成的。然后 URL 编码的查询字符串被附加到 Servlet URL 中，则整

个 URL 提交完成。URL 编码的查询字符串根据用户与可视部件之间的交互操作，将用户所选的值与可视部件的名称进行配对。例如，考虑前面的 HTML 代码段将用于显示按钮（标记为 AM 和 FM），如果用户选择 FM 按钮，则查询字符串包含 name=value 的配对操作为 broadcast=fm。因为在这种情况下，Servlet 将响应 HTTP 请求，因此 Servlet 应基于 HttpServlet 类。Servlet 应根据提交给它的查询字符串中的用户信息使用的 GET 或 POST 方法，而相应地使用 doGet()或 doPost()方法。

(3) 在<Servlet>标记中指定 Servlet。当使用<Servlet>标记来调用 Servlet 时，如同使用<FORM>标记一样，无须建立一个完整的 HTML 页面。作为替代，Servlet 的输出仅是 HTML 页面的一部分，且被动态嵌入到原始 HTML 页面中的其他静态文本中。所有这些都发生在服务器上，且发送给用户的仅是结果 HTML 页面。建议在 JSP 文件中使用<Servlet>标记，具体请参阅有关 JSP 技术。

原始 HTML 页面中包含<Servlet>和</Servlet>标记。Servlet 将在这两个标记中被调用，且 Servlet 的响应将覆盖这两个标记间的所有代码和标记本身。如果用户的浏览器可以看到 HTML 源文件，则用户将看不到<Servlet>和</Servlet>标记。要在 Domino Go WebServer 上使用该方法，需要启用服务器端。部分启用过程将会涉及到添加特殊文件类型 SHTML。当 Web 服务器接收到一个扩展名为 SHTML 的 Web 页面请求时，它将搜索<Servlet>和</Servlet>标记。对于所有支持该技术的 Web 服务器，如 WebSphere 应用服务器，将处理<Servlet>标记间的所有信息。

下列 HTML 代码段显示了如何使用该技术。

```
<Servlet name="myServlet"code="myServlet.class"CODEBASE="url"initparml=
                                "value">
  <PARAMNAME="parml"VALUE="value">
</Servlet >
```

使用 name 和 code 属性带来了使用上的灵活性，可以只使用其中一个属性，也可以同时使用两个。name 属性指定了 Servlet 的名称（使用 WebSphere 应用服务器管理器配置的），或不带.class 扩展名的 Servlet 类名，而 code 属性则指定了 Servlet 类名。使用 WebSphere 应用服务器时，建议同时指定 name 和 code，或当 name 指定了 Servlet 名称时，仅指定 name。如果仅指定了 code，则会创建一个 name=code 的 Servlet 实例。装入的 Servlet 将假设 Servlet 名称与 name 属性中指定的名称匹配。然后，其他 SHTML 文件可以成功地使用 name 属性来指定 Servlet 的名称，并调用已装入的 Servlet。name 的值可以直接在要调用 Servlet 的 URL 中使用。如果 name 和 code 都存在，且 name 指定了一个现有的 Servlet，则通常使用 name 中指定的 Servlet。由于 Servlet 创建了部分 HTML 文件，所以当创建 Servlet 时，将可能会使用 HttpServlet 的一个子类，并优先使用 doGet()方法（因为 get 方法是提供信息给 Servlet 的默认方法）。另一个选项是优先使用 service()方法。另外，codebase 是可选的，它指定装入 Servlet 的远程系统的 URL。应使用 WebSphere 应用服务器管理器来从 JAR 文件配置远程 Servlet 并装入系统。

在上述的标记示例中，initparml 是初始化参数名，value 是该参数的值，可以指定多个“名称—值”对的集合。利用 ServletConfig 对象(被传递到 Servlet 的 init()方法中)的 getInitParameterNames()和 getInitParameter()方法来查找参数名和参数值的字符串数组。在示例中，parml 是参数名，并在初始化 Servlet 后才被设置为某个值。因为只能通过使用“请求”对象的方法来使用以<PARAM>标记设置的参数，所以服务器必须调用 Servlet 的 service()

方法，以从用户处传递请求。要获得有关用户的请求信息，可以使用 `getParameterNames()`、`getParameter()` 和 `getParameterValues()` 方法。

初始化参数是可以持续使用的。假设一台客户机通过调用一个包含某些初始化参数的 SHTML 文件来调用 Servlet，并假设第二台客户机通过调用第二个 SHTML 文件来调用同一个 Servlet，且该 SHTML 中未指定任何初始化参数。那么第一次调用 Servlet 时所设置的初始化参数将一直可用，并且通过所有其他 SHTML 文件而调用的所有后继 Servlet 都不能更改该参数。直到 Servlet 调用了 `destroy()` 方法后，才能重新设置初始化参数。例如，如果另一个 SHTML 文件指定了一个不同的初始化参数值，虽然此时已装入了 Servlet，但该值仍将被忽略。

5. 在JSP文件中调用Servlet

如果某个类要成为 Servlet，则它应该从 `HttpServlet` 继承，根据数据通过 GET 还是 POST 发送，覆盖 `doGet()`、`doPost()` 方法之一或全部。`doGet()` 和 `doPost()` 方法都有两个参数，分别为 `HttpServletRequest` 类型和 `HttpServletResponse` 类型。`HttpServletRequest` 提供访问有关请求的信息的方法，如表单数据、HTTP 请求头，等等。`HttpServletResponse` 除了提供用于指定 HTTP 应答状态（200，404 等）、应答头（Content-Type，Set-Cookie 等）的方法之外，还提供了一个用于向客户端发送数据的 `PrintWriter`。对于简单的 Servlet 来说，它的大部分工作是通过 `println` 语句生成向客户端发送的页面。

注意 `doGet()` 和 `doPost()` 将抛出两个异常，因此必须在声明中包含它们。另外，还必须导入 `java.io` 包（要用到 `PrintWriter` 等类）、`javax.servlet` 包（要用到 `HttpServlet` 等类）以及 `javax.servlet.http` 包（要用到 `HttpServletRequest` 类和 `HttpServletResponse` 类）。

最后，`doGet()` 和 `doPost()` 这两个方法是由 `service` 方法调用的，有时可能需要直接覆盖 `service` 方法，例如，Servlet 要处理 GET 和 POST 两种请求时就是这样。

【例 4-13】 输出纯文本的简单 Servlet（文件名为 `HelloWorld.java`）。

```
package hall;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWorld extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("Hello World");
    }
}
```

6. Servlet的编译和安装

下面介绍编译包里面的类的两种调用方法。

一种方法是设置 `CLASSPATH`，使其指向实际存放 Servlet 的目录的上一级目录（Servlet 主目录），然后在该目录中按正常的方式编译。例如，如果 Servlet 的主目录是 `C:\JavaWebServer\servlets`，包的名字（即主目录下的子目录名字）是 `hall`，在 DOS 下，编译过程如下：

```
DOS> set CLASSPATH=C:\JavaWebServer\servlets;%CLASSPATH%
DOS> cd C:\JavaWebServer\servlets\hall
DOS> javac YourServlet.java
```

第二种编译包里面的 Servlet 的方法是进入 Servlet 主目录，执行“javac directory\YourServlet.java”（Windows）或者“javac directory/YourServlet.java”（Unix）。例如，再次假定 Servlet 主目录是 C:\JavaWebServer\servlets，包的名字是 hall，在 Windows 中编译过程如下：

```
DOS> cd C:\JavaWebServer\servlets
DOS> javac hall\YourServlet.java
```

注意：在 Windows 下，大多数 JDK 1.1 版本的 javac 要求目录名字后面加反斜杠(\)。JDK1.2 已经改正了这个问题，然而由于许多 Web 服务器仍旧使用 JDK 1.1，因此大量的 Servlet 开发者仍在使用 JDK 1.1。

7. 运行Servlet

在 Java Web Server 下，Servlet 应该放到 JWS 安装目录的 servlets 子目录下，而调用 Servlet 的 URL 是 http://host/servlet/ServletName。注意子目录的名字是 servlets（带“s”），而 URL 使用的是“servlet”。由于 HelloWorld Servlet 放入包 hall，因此调用它的 URL 应该是 http://host/servlet/hall.HelloWorld。在其他的服务器上，安装和调用 Servlet 的方法可能略有不同。

下面介绍一个输出 HTML 的 Servlet。

大多数 Servlet 都输出 HTML。要输出 HTML 还有两个额外的步骤要做：告诉浏览器接下来发送的是 HTML；修改 println 语句构造出合法的 HTML 页面。

第一步通过设置 Content-Type（内容类型）应答头完成。一般地，应答头可以通过 HttpServletResponse 的 setHeader 方法设置，但由于设置内容类型是一个很频繁的操作，因此 Servlet API 提供了一个专用的方法 setContentType。注意设置应答头应该在通过 PrintWriter 发送内容之前进行。

【例 4-14】一个设置应答头的实例（文件名为 HelloWWW.java）。

```
package hall;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWWW extends HttpServlet {
    public void doGet(HttpServletRequest request,HttpServletResponse response)
throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" +
        \"Transitional//EN\">\n\" + "<HTML>\n\" +
        "<HEAD><TITLE>Hello WWW</TITLE></HEAD>\n\" +
        "<BODY>\n\" +
        "<H1>Hello WWW</H1>\n\" +
        "</BODY></HTML>\"=";
```

JSP 并没有增加任何本质上不能用 Servlet 实现的功能。但是，在 JSP 中编写静态

HTML 更加方便，不必再用 `println` 语句来输出每一行 HTML 代码。更重要的是，借助内容和外观的分离，页面制作中不同性质的任务很容易分开。例如，由页面设计专家进行 HTML 设计，同时留出供 Servlet 程序员插入动态内容的空间。

4.2.2 开发Servlet应用

这是由 Apache 提供的一个 Servlet 应用实例，从例子中可以学到使用 Servlet 的一般方法。

【例 4-15】 Apache 提供的一个使用 Servlet 的示例程序。

这段程序是 Apache 提供的一个测试 Servlet 应用的一个标准范例。首先对下面这段 Servlet 进行编译：

```
/**
 * Copyright (c) 1999 The Apache Software Foundation. All rights
 * reserved.
 */
import javax.servlet.*;
import javax.servlet.http.*;

public class servletToJsp extends HttpServlet {

    public void doGet (HttpServletRequest request,
                      HttpServletResponse response) {

        try {
            // Set the attribute and Forward to hello.jsp
            request.setAttribute ("servletName", "servletToJsp");
            getServletConfig().getServletContext().getRequestDispatcher
                ("/jsp/jsptoserv/hello.jsp").forward(request, response);
        } catch (Exception ex) {
            ex.printStackTrace ();
        }
    }
}
```

以下是使用 JSP 调用上面编译好的 Servlet 的 JSP 文件的代码：

```
<html>
<!--
    Copyright (c) 1999 The Apache Software Foundation. All rights
    reserved.
-->
<body bgcolor="white">

<!-- Forward to a servlet -->
<jsp:forward page="/servlet/servletToJsp" />
</html>
```

4.2.3 Servlet与JSP、JavaBean协同工作

在使用 JSP 技术开发网站时，并不强调使用 Servlet。这是为什么呢？虽然 Servlet 非常适合服务器端的处理和编程，但是如果用 Servlet 处理大量的 HTML 文本，那么将是一件极其烦琐的事情。这种事情更适合计算机去做，否则，就是浪费程序员的体力。所以 Servlet 更适合处理后端的事务，前端的效果用 JSP 来实现更为合适。

如图 4-5 所示，Servlet 与 JSP、JavaBean 的协同工作模式是一种面向动态内容的 MVC (Model 模型、Controller 控制器、View 视图) 框架实现，结合了 Servlet (Controller 控制器) 和 JSP (View 视图) 技术。它利用两种技术原有的优点，采用 JSP 来表现页面，采用 Servlet 来完成大量的处理，Servlet 扮演一个控制者的角色，并负责响应客户请求。接着，Servlet 创建 JSP 需要的 Bean (Model 模型) 和对象，再根据用户的行为，决定将哪个 JSP 页面发送给用户。特别要注意的是，JSP 页面中没有任何商业处理逻辑，它只是简单地检索 Servlet 已创建的 Beans 或者对象，再将动态内容插入预定义的模板。在此模式中，JSP 和 Servlet 可以在功能上最大幅度地分开。正确使用此模式，将会有有一个中心化的控制器 (Servlet)，以及只完成显示的 JSP 页面。

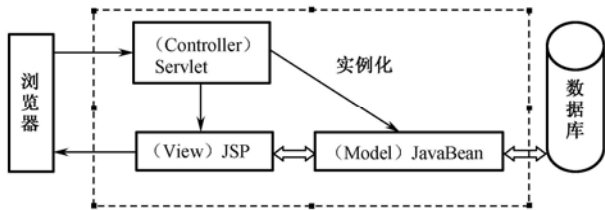


图 4-5 Servlet 与 JSP、JavaBean 协同工作示意图

从开发的观点来看，这一模式具有清晰的页面表现，清楚的开发者角色划分，可以充分利用开发小组中的界面设计人员。事实上，越是复杂的项目，使用此模式的好处就越突出。例如，Struts 技术框架（见 5.2 Struts2 框架）就是此模式最好的实现。

在实现 MVC 框架的 Web 应用时，通常都先经过 Servlet 的处理，然后再把处理的结果传给 JSP 页面以显示给用户，这时，Servlet 就需要向 JSP 页面传输数据。

前面已经说过，可以把需要传输到下个页面的数据绑定到 HttpSession、ServletContext 和 HttpServletRequest 对象，然后再调用 forward () 方法进行数据传递。但当需要传输的数据过多时，直接采用这种方法显得有些零散，不利于开发管理。其实，可以把一些有关联的、描述同一事物的数据捆绑成一个数据对象，然后再进行传递，这样，可以通过 JavaBean 的方式来实现。

JavaBean 是一种封装属性和方法的类，可以用来存储数据和提供某些特殊的功能。

在本例中，JavaBean 文件 User.java 用于存储用户的信息。

【例 4-16】 如何使用 JavaBean 由 Servlet 向 JSP 传输数据 (User.java 源代码)。

```
//User.java
package myBean;
public class User {
    String name;
    int age;
    public User() { }
```

```

    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}

```

该 **JavaBean** 有两个属性，即姓名和年龄，每个属性都有对应的 `setxxx()` 方法和 `getxxx()` 方法。

文件 `UseBeanServlet.java` 使用 **JavaBean** 保存数据，将 **Bean** 的实例绑定到 `HttpSession` 对象上，然后调用 `forward()` 方法重定向到 `userBean.jsp`。

`UseBeanServlet.java` 内容如下：

```

//UseBeanServlet.java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import myBean.User;

public class UseBeanServlet extends HttpServlet {
    public void doGet (HttpServletRequest request,HttpServletResponse
                        response)
        throws ServletException,IOException {
        User usr = new User();
        usr.setAge(18);
        usr.setName("somebody");
        HttpSession session = request.getSession();
        session.setAttribute("user",usr);
        getServletConfig().getServletContext().getRequestDispatcher
            ("/usebean.jsp").forward(request,response);
    }
}

```

文件 `usebean.jsp` 从会话对象上取出在 `UseBeanServlet.java` 绑定的 `User` 对象，然后对其信息输出。`usebean.jsp` 文件内容如下：

```

<%@ page contentType = "text/html;charset = GBK" %>
<%@ page import = "myBean.User" %>
<html>
<head>
<title>使用 JavaBean 从 Servlet 传递数据到 JSP</title>
</head>
<body>
<%
    User user = (User)session.getAttribute( "User" );
    out.print( "姓名: "+user.getName()+ "<BR>年龄: "+user.getAge() );
%>
</body>
</html>

```

对 `UseBeanServlet.class` 这个 `Servlet` 做简单配置，运行该 `Servlet` 就可以得到所要的结果。

4.3 用JSP访问数据库

数据库连接对动态网站来说是最为重要的部分。如何获取数据、增加数据、删除数据以及对数据库进行管理，是每个程序开发者必须面对的问题。

4.3.1 用JSP访问SQL Server数据库

SQL Server 2000 JDBC Driver 是最好的 Type 4 JDBC 驱动程序，它提供了面向企业的、与 Java 环境的高度可靠、高度可伸缩的连通性。SQL Server 2000 JDBC Driver 为所有 Java 小程序(Java-enabled Applet)、应用程序或者应用程序服务器提供了 JDBC 访问能力。它跨越 Internet 和 Intranet 提供了对 SQL Server 2000 的高性能点对点访问和 n 层 (n-tier) 访问。该驱动程序针对 Java 环境进行了优化，使用户可以将 Java 技术与现有的系统相结合，以扩展现有系统的功能和性能。可以直接到微软公司的以下站点下载 SQL Server 2000 JDBC Driver 并安装。

<http://www.microsoft.com/china/sql/downloads/2000/jdbc.asp>

SQL Server 2000 JDBC Driver 是一个遵从 JDBC 2.0 规范的驱动程序。它还支持 JDBC 2.0 Optional Package 的一个子集，该子集提供了以下一些功能（详细信息参见文档）：

- (1) Java 命名目录接口 (JNDI)，用于命名数据源；
- (2) 连接池 (Connection Pooling)；
- (3) SQL Server 2000 JDBC Driver 支持的 SQL Server；
- (4) SQL Server 2000 JDBC Driver 支持的 JDK。

下面是一个使用 SQL Server 的 JSP 实例。

在练习这些代码的时候，要在数据库里建一个表 `test`，有两个字段如 `test1`，`test2`，可以用下面的 SQL 语句建立以下一个数据库：

```
create table test(test1 varchar(20),test2 varchar(20))
```

然后向这个表写入一条测试记录，接着进行 JSP 和数据库连接应用。

【例 4-17】 JSP 连接数据库（文件名为 `testsqlserver.jsp`）。

```
<%@ page contentType="text/html; charset=gb2312"% >
<%@ page import="java.sql.*"% >
<html >
<body >
<%Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver").newInstance();
String url="jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=pubs";
//pubs 为你的数据库的名称
String user="sa";
String password="";
Connection conn= DriverManager.getConnection(url,user,password);
Statement stmt=conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```



```
String sql="select * from test";
ResultSet rs=stmt.executeQuery(sql);
while(rs.next()) {% >
您的第一个字段内容为: < %=rs.getString(1)% >
您的第二个字段内容为: < %=rs.getString(2)% >
< %}% >
< %out.print("数据库操作成功, 恭喜您");% >
< %rs.close();
stmt.close();
conn.close();
% >
< /body >
< /html >
```

4.3.2 JSP用JavaBean操纵数据库

在 JSP 技术应用中, 常应用 JavaBean 来操作数据库。下面通过一个实例来介绍用 JavaBean 操纵数据库的基本方法。

首先建立一个数据库 Customers.mdb, 其中表 Customers 有字段 id (自动增量型, 并设为主关键字)、name (文本型, 长度 10)、address (文本型, 长度 30)、info (备注型)。在 Control Panel (控制面板) 的 ODBC Datasource 模块中加入 System DSN, 取名 Customers, 并指向 Customers.mdb。创建一个 JavaBeans, 命名为 DBconn.java, 并保存在支持 JSP 的 Web 服务器的默认文档根目录下。DBconn.java 主要用于封装与数据库的连接操作。

【例 4-18】用 JavaBean 操纵数据库。

```
public class DBconn {
    String DBDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
    String ConnStr = "jdbc:odbc:Customers";
    Connection conn = null;
    ResultSet rs = null;
    public DBconn {
        try {
            Class.forName(DBDriver);
            //加载数据库驱动程序
        }
        catch(java.lang.ClassNotFoundException e) {
            System.err.println("DBconn (): " + e.getMessage());
        }
    }
    public ResultSet executeQuery(String sql) {
        rs = null;
        try {
            conn = DriverManager.getConnection(ConnStr);
            //与 DBMS 建立连接
            Statement stmt = conn.createStatement();
```

```

        rs = stmt.executeQuery(sql);
    }
    catch(SQLException ex) {
        System.err.println("aq.executeQuery: " + ex.getMessage());
    }
    return rs;
}
}

```

DBconn.java 编辑好后，在 DOS 状态下，进而用 JDK 的 javac 命令编译 DBconn.java 形成相应的 class 文件。接着建立 Customers.jsp 文件，在 JSP 中调用以上编译好的 JavaBeans，其内容如下：

```

<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=gb2312">
    <title>客户信息调查</title>
</head>
<body>
    <p><b>客户信息调查</b></p>
    <%@ page language="java" import="java.sql.*" %>
    <jsp:useBean id="DBconn1" scope="page" class="DBconn" />
    <%
        ResultSet RS = DBconn1.executeQuery("SELECT * FROM Customers");
        while (RS.next()) {
            out.print("<LI>" + RS.getString("name") + "</LI>");
            out.print("<LI>" + RS.getString("address") + "</LI>");
            out.print("<LI>" + RS.getString("info") + "</LI>");
        }
        RS.close();
    %>
</body>
</html>

```

本例中，在<jsp:useBean>标记内定义了几个属性，其中 id 是整个 JSP 页面内该 Bean 的标识、scope 属性定义了该 Bean 的生存时间、class 属性说明了该 Bean 的类文件。

4.4 JSTL标准标签库技术

4.4.1 JSTL及其操作实现

本章的大部分内容都集中在如何创建自己的定制操作和标签方面。但是实际中并不限于只使用自己创建的标签，也可以利用任何其他的标签库。目前已经有很多标签库，例如，本节将介绍使用 JSP 标准标签库，也就是 JSTL（JSP Standard Tag Library）。

JSTL 是从许多开发者创建的标签库发展而来的，JSTL 对常用的操作进行了标准化。例如，本来使用标准标签库中某一个实现，如果想要转换到另外一个标准标签库，只要把.jar

文件添加到应用中，并修改 Web.xml 文件以建立 taglib-uri 和新的 TLD 的映射关系即可。本节将探讨 JSTL 实现、包括 JSTL 的一些操作以及如何使用 JSTL。

1. 准备知识——标签库描述文件（TLD）

在创建了一个标签的一个或多个实现类后，需要告知容器这个应用程序的 JSP 页面用的是哪个标签处理程序。这一步是通过标签库描述文件（TLD）来完成的。TLD 文件是 XML 兼容的文档，它包含标签库中的标签处理程序类的信息。JSP 页面中的标签必须符合 TLD 描述的约束。如果没有正确按照 TLD 描述来使用这些标签，那么将会出现解释错误。

JSP 2.0 的 TLD 用 <taglib> 元素描述与该标签库有关的信息，参见例 4-19。

【例 4-19】 TLD 的例子。

```
<? xml version = "1.0" encoding = "UTF-8"?>

<taglib xmlns= "http://java.sun.com/xml/ns/javaee"
        xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation = "http://java.sun.com/xml/ns/javaee/web-
        jsptaglibrary_2_1.xsd" version = "2.1">
<tlib-version>1.0</tlib-version>
<short-name>aprèss</short-name>
<tag>
    <name>example</name>
    <tag-class>com..aprèss.faq.Example</tag-class>
    <body-content>empty</body-content>
    <variable>
        <name-given>script1</name-given>
    </variable>
    <variable>
        <name-from-attribute>attr1</name-from-attribute>
    </variable>
    <attribute>
        <name>attr1</name>
        <required>yes</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
    < attribute >
        <name>attr2</name>
        <required>no</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
</tag>
</taglib>
```

这里的 <taglib> 元素可以包含一些子元素，其中有一些必需的子元素，如表 4-1 所示。

表 4-1 <taglib>必需的子元素

元 素	含 义
tlib-version	标签库的版本号
short-name	一个简单的默认名称，可以用做 taglib 指令的前缀值
tag	标签处理程序的有关信息

表中的<tag>元素还有若干子元素，其中 name 和 tag-class 是必填的两个。另外还会常用到<tag>元素的其他可选子元素，如表 4-2 所示。

表 4-2 <tag>的子元素

元 素	含 义
name	标签处理程序的名字（必填）
tag-class	标签处理类的完整有效的类名（必填）
body-content	表示该标签是否有标签体部分的内容。有效值是 tagdependent、scriptless 或 empty。默认值是 scriptless。如果值为 empty，那么这个标签不允许有标签体部分
variable	定义由标签处理程序创建的脚本变量，且对于页面余下部分有效。这个元素必须包括以下两个子元素之一：name-given 或 name-from-attribute。如果使用了 name-given，那么这个元素值直接定义了一个名称，其他 JSP 元素可以用它来访问已创建好的脚本变量。如果选用 name-from-attribute，那么属性值和这个元素所给定的名称一起定义了脚本变量名
attribute	定义标签的属性。这个元素有三个子元素：name、required 和 rtexprvalue。name 元素是该属性的名称。required 是一个可选元素，它的值必须是 true、false、yes 或 no 之一，它表示这个属性是必需的还是可选的，其默认值是 false，表示属性是可选的。rtexprvalue 也是一个可选元素，其值必需是 true、false、yes 或 no 之一。默认值 false 表示这个属性只能在编译时用一个静态值来设置；如果这个元素值为 true 或 yes，那么表示这个属性可以用运行时表达式来设置

例 4-19 所示的 TLD 是开发者基于 JSP 2.0 的规范编写的一个标签库描述文件，其版本为 1.0。它建议库中标签的前缀为 apress。但是需要注意，页面开发者可以用任何他们所期望用的前缀。这样也可以避免各个标签库都用建议的前缀而产生冲突。

这个 TLD 定义了一个名为 example 的标签。这个标签的标签处理类是 com.apress.faq.Example。因为<body-content>元素的值是 empty，所以这个标签没有标签体部分。

这个标签创建了两个对象，在页面中可以作为脚本变量来用。JSP 页面用名字 scriptl(由<name-given>元素指定)访问第一个对象，而用这个标签的 attr1 属性设定的名称访问第二个对象。

这个标签包含两个属性。属性 attr1 是必需的，并且可以由运行时表达式来设置。属性 attr2 是可选的，它不能由运行时表达式来设置。

如果 rtexprvalue 的值为 true，那么这个属性可以采用 \${expr}格式的 EL 表达式。如果 tag 元素还包含<deferred-value/>或<deferred-method/>二者之一的空元素，那么这个属性可以用#{expr}格式的 EL 表达式来设置。关于如何用请求时表达式来设置属性值，请参考 JSP 规范。

2. 获得一个JSTL实现

如果想用 JSTL，那么可以从 Jakarta 的 JSTL 项目中获得一个实现，项目网址为 <http://jakarta.apache.org/taglibs/index.html>。例如，版本 JSTL 1.1.2 等运行在 Tomcat 5 环境下。

JSTL 用起来很简单：

- (1) 将发布的压缩包中的文件释放到应用中。`.jar` 文件包含了标签处理程序，把它放到 `/WEB-INF/lib` 下，然后把其 `TLD` 文件放到 `/WEB-INF/` 下。
- (2) 修改 `Web.xml` 文件，添加 `taglib-uri` 元素和 `TLD` 位置的映射关系。
- (3) 在使用 `JSTL` 标签的页面中增加 `taglib` 指令。

3. JSTL中的操作

JSTL 的标签分为四类：核心操作（`c.tld`）、XML 处理（`x.tld`）、国际化格式处理（`fmt.tld`）、关系数据库访问（`sql.tld`）。

如果使用的 Web 容器支持 JSP 2.0（或更新版本），并且使用的是 JSTL 1.1.2（或更新版本），那么将会用到列出的 TLD 文件。因为 Tomcat 5 是 JSP 2.1 的参考实现，所以将用到 JSTL 1.1.2 版本的 TLD 文件。首先应当把这些 TLD 文件复制到 Web 应用的 `tld` 目录下。

如果用的是 1.1.2 发布版，那么需要注意有一些额外的 TLD 文件。例如，相对于 `c.tld` 文件，有 `c-1.0.tld` 和 `c-1_0-rt.tld` 文件。另外对于 `x.tld`、`fmt.tld` 和 `sql.tld` 文件也有类似命名的文件。这些 TLD 文件向前兼容 JSTL 1.0 版本（因此文件名里包含 `1_0`）。

如果使用的 Web 容器下兼容 JSP 2.0（或更新版本），那么需要用这些 JSTL 1.0 版本的 TLD 文件。如果页面用到了 Java 脚本表达式（`<% ! %>`、`<% = %>` 或 `<% %>`），那么要用 `rt` 版本的 TLD。缩写 `rt` 表示 `rtexpvalue`，是指运行时（或请求时）表达式的值。JSP 规范中常使用运行时（`runtime`）和请求时（`request-time`）这两个词。如果页面里用到了 EL 表达式，那么应该选用另外的一个版本。如果页面里既用到 Java 脚本又用了 EL，那么这两个 TLD 都需要用到。在同一个页面里，可以混合使用不同标签库的标签。但是要记住这种情况只适用于容器不支持 JSP 2.0（或更新的版本）的情况。例如，Java EE 参考实现用 Tomcat 5，它支持 JSP 2.1 的 Web 容器。

(1) 核心操作。核心操作用来为标签处理程序提供变量操作、错误处理、执行条件判断以及循环和迭代。核心操作中包含通用的操作为变量处理和错误处理提供了支持。表 4-3 描述了通用的操作以及如何使用它们。

表 4-3 JSTL 核心分类中的通用操作

标 签	含 义
<code><c:out value = "" default = ""></code>	把值（value）发送到响应（response）流，可以指定一个默认值。如果用一个 EL 表达式设置了此值的属性，而该表达式的值是 <code>null</code> ，那么将输出这个默认值
<code><c:set var = "" value= ""></code>	通过 <code>var</code> 和它的值（value）来设置一个 JSP 作用域的变量值
<code><c:set target = ""property= ""value= ""></code>	把 <code>JavaBean</code> 或 <code>Map</code> 对象的属性设置为某个值
<code><c:remove var = "" scope = ""></code>	从给定的作用域中删除由 <code>var</code> 指定的对象。其中 <code>scope</code> 属性是可选项。如果没有给定 <code>scope</code> ，那么将按照 <code>page</code> 、 <code>request</code> 、 <code>session</code> 、 <code>application</code> 的顺序来搜索作用域，直到找到该对象或搜索完所有域为止。如果给定了 <code>scope</code> ，那么只有对象在指定的作用域中才能删除。如果最终没有找到该对象，那么将抛出一个异常
<code><c:catch var = ""></code>	可能抛出异常的代码块。如果出现异常，那么这个代码块会终止，但是异常不会传递下去，可以通过由 <code>var</code> 命名的变量引用这个异常

条件操作用于判断表达式并且根据判断的结果来执行标签，如表 4-4 所示。

表 4-4 JSTL 核心分类中的条件操作

标 签	含 义
<c:if test = ""var = "">	用法类似于标准 Java 的 if 语句块。属性 var 是可选的；如果有 var 属性，那么判断的结果将赋予变量 var。如果判断结果为 true,那么将执行标签；如果为 false，将不执行标签
<c:choose>	类似于 Java 的 if...elseif...else 语句块。<c:choose>操作在语句和结尾处
<c:when test = "">	判断每个<c:when test = "">标签的值；如果第一个判断等于 true，那么将执行 tag
<c:otherwise>	如果所有<c:when>标签都不是 true，那么将执行<c:otherwise>标签

如表 4-5 所示为迭代操作，这些操作用于循环访问一组值。

表 4-5 JSTL 核心分类里的迭代操作

标 签	含 义
<c:forEach var = ""items = "">	迭代 items 集合中每个条目。用 var 引用每个条目。当 items 是一个 Map 时，则用 var.value 引用条目的值
<c:forEach var= ""begin= ""end= ""step= "">	for 循环的标签。step 属性是可选的
<c:forTokens items= ""delims= "">	在 items 字符串中迭代访问每个单词

4. SQL操作

页面的开发者可以用 JSTL 的 SQL 操作来执行数据库查询、访问查询结果和执行插入、更新和删除操作。其中之一是<sql:query>：

```
<sql:query var= ""dataSource= ""> SQL 命令</ sql:query>
```

这个操作对 dataSource 属性设定的数据库进行查询。相应的查询语句是在标签体中设定的。然后可以用 var.rows 访问查询的结果，并且用<c:forEach>标签迭代 rows 集合。

其中的 dataSource 属性有两种标识数据库的方式：用 JDBC URL 访问数据库，或者用 Java 命名和目录接口（Java Naming and Pireceory Interface, JNDI）数据源名称来查询数据库。

4.4.2 在JSP中使用JSTL

下面这个 JSP 用于说明 JSTL 的各种用法。从 Jakarta 的网站 <http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html> 上下载 JSTL 1.1 发布版本。把 TLD 文件 c.tld 和 fmt.tld 解压缩到本章例子使用的/WEB-INF/tlds 目录下，然后再把 standard.jar 和 jstl.jar 两个文件解压缩到/WEB-INF/lib 目录下。

建立例 4-20 中的 EL-1.jsp 文件，把这个文件保存在/WEB-INF/目录下。

【例 4-20】EL_1.jsp。

```
<%@taglib uri = "http://java.sun.com/jsp/jstl/core" prefix = "c"%>
<%@taglib uri = "http://java.sun.com/jsp/jstl/fmt" prefix = "fmt"%>
<html>
<head>
<title>JSTL Q2</title>
</head>

<body>
```

<h1>JSTL Question 2 </h1>

<h2>How do I use the JSTL?</h2>

```
<jsp:useBean id = "questions" class = "com.aprèss.faq.Questions"
            scope = "page">
```

```
<jsp:setproperty name = "questions" property = "topic" value = "EL"/>
```

```
</jsp:useBean>
```

```
<table border = "1">
```

```
<!-- the literal JSTL tag will be in left column of table -->
```

```
<!-- the evaluated JSTL tag will be in right column of table -->
```

```
<tr><th>tag</th><th>result</th></tr>
```

```
<!-- this tag uses c:out to send the value of an EL to the response -->
```

```
<tr><td>&lt; c:out value = "${ '}'questions.topic}"/&gt;</td>
```

```
<td><c:out value = "${questions.topic}"/></td>
```

```
</tr>
```

```
<!-- this tag uses c:set to set the property of a JavaBean -->
```

```
<c:set target = "${questions}" property = "topic" value = "JSTL"/>
```

```
<tr>
```

```
<td>&lt;c:set target = "${'}'questions}" property = "topic"
        value = "JSTL"/&gt;
```

```
</td>
```

```
<td><c:out value = "${question.topic}"/></td>
```

```
</tr>
```

```
<!-- this tag uses c:if to determine whether to create another row -->
```

```
<c:if test = "${questions.topic = 'EL'}">
```

```
<tr><td>This row will not be created</td>
```

```
<td></td>
```

```
</tr>
```

```
</c:if>
```

```
<c:if test = "${question.topic = 'JSTL'}">
```

```
<tr><td>This row was created because the c:if tag result was true</td>
```

```
<td></td>
```

```
</tr>
```

```
</c:if>
```

```
</table>
```

<h2>Multiplication table,1 - 5</h2>

```
<!-- use the forEach tag to create a table -->
```

```
<table border = "1">
```

```

<tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr>
<c:forEach var = "i" begin = "1" end = "5"
  <tr><td><c:out value = "${i}" /></td>
    <c:forEach var = "j" begin = "1" end = "5">
      <td><c:out value = "${i*j}" /></td>
    </c:forEach>
  </tr>
</c:forEach>
</table>

```

<h2>Formatting numbers</h2>

```

<br>&lt;fmt:formatNumber value = "23.456" type = "number"&gt; results in
  <fmt:formatNumber value = "23.456" type = "number"/>
<br>&lt;fmt:formatNumber type = "currency"&gt;23.456
  &lt;/fmt:formatNumber&gt; results in <fmt:formatNumber
  type = "currency">23.456</fmt:formatNumber>
<br>&lt;fmt:formatNumber value = ".23456" type = "percent"&gt; results
  in< fmt:formatNumber value = ".23456" type = "percent"/>
<br>&lt;fmt:formatNumber value = ".23456" type = "percent"
  minFractionDigits = "2"&gt; results in <fmt:formatNumber
  value = ".23456" type = "percent" minFractionDigits = "2"/>

```

</body>

</html>

如果用的是 Tomcat, 那么需要修改 web.xml 文件, 如例 4-20。如果用的是 Java EE 部署工具, 则通过这个部署工具来设置 taglib 的映射关系。

【例 4-21】Jsp_Ex09 应用的 web.xml 文件。

```

<?xml version = "1.0" encoding = "ISO-8859-1"?>

```

```

<web-app xmlns = "http://java.sun.com/xml/ns/javaee"
  xmlns:xsi = "http://www.w3.org//2001/XMLSchema-instance"
  xsi:schemaLocation= "http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  version = "2.5" >

  <display-name>Jsp_Ex09 - JSP Standard Tag Library</display-name>

  <welcome-file-list>
    <welcome-file>welcome.jsp</welcome-file>
  </welcome-file-list>

  <jsp-config>
    <taglib>
<taglib-uri>/simplequestions</taglib-uri>
<taglib-location>/WEB-INF/tlds/simplefaq.tld</taglib-location>
    </taglib>

```



```

    <taglib>
<taglib-uri>http://java.sun.com/jsp/jstl/core</taglib-uri>
<taglib-location>/WEB-INF/tlds/c.tld</taglib-location>
    </taglib>

    <taglib>
<taglib-uri>http://java.sun.com/jsp/jstl/fmt</taglib-uri>
<taglib-location>/WEB-INF/tlds/fmt.tld</taglib-location>
    </taglib>
    <jsp-property-group>
<url-pattern>*.jsp</url-pattern>
<el-ignored>>false</el-ignored>
<scripting-invalid>>true</scripting-invalid>
    </jsp-property-group>
</jsp-config>
</web-app>

```

这个web.xml中列出了需要用到的全部标签。部署这些新文件，然后在浏览器中输入URL地址 http://localhost:8080/Jsp_Ex09/welcome.jsp。单击EL_1 的链接，将会看到一个页面。这个页面演示了一些JSTL标签。前面已经多次讲过TLD和web.xml，所以这里不再赘述。EL_1.jsp文件一开始先引入这些标签库：

```

<%@ taglib uri = "http://java.sun.com/jsp/jstl/core" prefix = "c" %>
<%@ taglib uri = "http://java.sun.com/jsp/jstl/format" prefix = "fmt"%>

```

我们所用的每个标签库都需要一个 taglib 条目。EL_1.jsp 使用了一些核心和格式化库，其中每个库都有一个 taglib 条目。按照 JSTL 的建议，taglib 条目设定核心库的前缀为 c，格式化库的前缀为 fmt。但是前面讲过，可以把前缀设置为任何值；前缀主要是由页面设计者来设置，而不是由标签库实现者来规定。

正如前面所提到的，如果 Web 容器只支持 JSP 1.2，那么只能用 JSTL 1.0 版本的 TLD 文件，这样 JSP 页面中用到的每个库都有两个 taglib 条目。另外，taglib 元素的 URL 也有些不同。如果 JSP 页面用了 JSP 脚本表达式（如脚本程序或声明），那么要用 TLD 的 RT 版本，如 ftm1_0-rt.tld。因此，如果把例 4-20 部署到 JSP 1.2 的容器中，那么 taglib 条目就如下编写：

```

<%@ taglib uri = "http://java.sun.com/jstl /core" prefix = "c" %>
<%@ taglib uri = "http://java.sun.com/jstl /format" prefix = "fmt" %>
如果向页面中添加任何 Java 脚本程序、表达式或声明，那么需要增加下面的 taglib 条目：
<%@ taglib uri = "http://java.sun.com/jstl /core_rt" prefix = "c_rt" %>
<%@ taglib uri = "http://java.sun.com/jstl /format_rt" prefix = "fmt_rt" %>

```

比较上面的 URI 和例 4-20 的 URI，可以发现 JSTL 1.0 的 URI 不同于 JSTL 1.1 的 URI。JSTL 1.1 的 URI 在路径里增加了名称 jsp。

这个页面接下来创建了一个 Questions 类的 JavaBean，并且输出 topic 属性的值。然后把 topic 属性设置为另外一个值并且打印输出。接着用两个<c:if>标签来控制是否创建表格的下一行。

这个页面后面的部分用嵌套的标签<c:forEach>创建一个二维表格，并且填入从 1 到 5

的矩阵行列相乘的值。

这个例子同时也涉及 JSTL 中其他一些标签的用法，相关的具体说明请参考 JSTL 规范（可以从 Jakarta 网站和 java.sun.com 获得）。

当然，JSTL 不是唯一的标签库，还有很多商品用的和免费的标签库可用。下面是部分简明列表：

- **Struts**：Struts 标签库可用于构建“模型—视图—控制器（MVC）”的应用（见 5.2.6 Struts2 的标签库）。
- **JNDI**：这个标签库出自 Jakarta。如果想使用 Java 命名和目录接口（JNDI）的 API，则可以用这个标签库。在实际的 Web 应用中常常会使用 JNDI 来查找应用资源。详细的 JNDI 信息可以从 <http://java.sun.com/products/jndi/> 找到。这个标签库的网址是 <http://jakarta.apache.org/taglibs/doc/jndi-doc/intro.html>。
- **JSPTags.com**：这不单是一个标签库，而是标签库的集中地，网址是 <http://jsptags.com/tags/index.jsp>。如果这里没有你想要的标签，则需要自己制作标签。

从这些资源出发，应该很容易找到应用所需的功能的标签库。一旦找到合适的标签库，可按照下面的步骤使用它们：

- （1）解压缩库文件，把 .jar 文件放到应用相应的目录下。
- （2）修改 web.xml 文件，建立 URI 到其库位置的映射关系。
- （3）在用到 JSTL 标签的页面中增加 taglib 指令，然后使用这些标签。

此外，如果没有找到与你所需功能相匹配的标签库，那么可以利用所学的简单标签处理程序和标准标签处理程序的相关知识来编写自己的标签库。

4.5 JSF 技术

JavaServer Faces (JSF) 是 Java EE 领域中一种比较新的技术，它用于简化 Web 应用的开发，包括建立用户界面组件和页面，以及利用这些组件与业务对象建立关联，另外它还可以自动实现 bean 的使用和页面导航。

JSF 技术吸收了 JSP、Servlet 和其他 Web 应用框架的经验，最重要的是它建立在 Apache 的 Struts 项目基础上。其实这并不奇怪，因为 Struts 的创建者也是 JSF 规范的工程师（如果你热衷于 Struts，那么可以考虑结合使用 Struts 和 JSF；请参考 <http://struts.apache.org>）。

4.5.1 JSF 及其安装

关于 JSF 如何帮助 Web 应用开发者创建用户界面（UI），JSF 规范列出了如下的几点：

- ① 通过一套可复用的 UI 组件很方便地构建 UI；
- ② 简化了应用数据从 UI 转出或转入；
- ③ 在不同请求之间管理 UI 的状态；
- ④ 易于建立和复用定制的 UI 组件。

因为有了可复用的 UI 组件，所以开发 UI 就变得容易多了。这些 UI 组件相应有许多类，它们是 JSF 规范和实现的组成部分。在应用中使用这些组件很简单，而不必担心页面布局的语法。JSF 通过定制的 render（绘制）工具和 render 过程可以把这些组件转换为恰当的页面布局代码。JSF 的实现包含一个 HTML 的默认的 render 工具，不同客户端的 render 工具也许不同，但可以采用 render 相同的 JSF 代码。也就是说，不同的客户端系统可以具有

相同的 JSF 代码，只需要不同的 render 来为每个客户端定制各自的 UI。

通过 JSF 的实现来处理数据转移的机制，从而简化了应用数据从 UI 的转出或转入。利用 JSF 很容易地设定把哪些数据转移到什么地方，并且 JSF 实现了从 UI 对象到业务的数据迁移，反之亦然。JSF 的实现自动管理着不同应用请求之间的状态，所以不需要管理或实现任何会话处理程序。

同样，JSF 还提供了一种用于事件处理的简单管理方式，可以设定被关注的事件和设定用于处理这些事件的业务对象或类，事件产生后，JSF 的实现将调用相应的方法来处理。JSF 的事件处理模型类似于其他 UI 框架所用的模型，如 Java Swing。这样，一个事件可以有多个事件监听者（listener）对其进行响应。

最后，因为基于可复用组件技术来设计 JSF，所以很方便在 JSF 的应用中使用自己开发的组件或集成第三方的组件。

提示：与其他所有Java EE技术类似，JSF的详细信息可以参见JSF规范（<http://java.sun.com/j2ee/javaxserverfaces>）。

1. JSF和其他Java EE技术之间的关系

在 Java EE 中，如 JSP 页面、Servlet 和 EJB 都是独立的技术，可以只用 EJB、Servlet 或 JSP 页面中的一种技术来创建一个应用。

但是 JSF 则不然，它只是一种辅助（supporting）技术，需要结合 JSP 页面、Servlet 或 EJB 来使用它。

JSF 中主要的设计模式是“模型—视图—控制器（MVC）”。前面几章曾学习了用 JSP 来创建 UI。利用 JSP 很容易就可以创建出并管理 Web 应用的视图组件，同样也可用 Servlet 或 EJB 来创建 UI。现在，把 JSF 技术和这些技术结合起来使用，把模型、视图和控制器集成在一起，这样创建 UI 比过去更简单。在 Web 应用的开发中，JSF 运用了一种基于组件模型，它类似于单机版 GUI 应用的模型。

为了结合 Servlet 或 EJB 来使用 JSF，需要直接用组件来建立 JSF，而我们将重点讨论如何在 JSP 中使用 JSF。JSF 的实现包含了一个由许多定制标签组成的标签库，使用这个标签库，就可以很容易创建出 JSF 应用。

2. JSF的安装

为了运行本节的例子，需要下载和安装一份 JSF 的参考实现、一套 JSP 标准标签库（JSTL）的参考实现和一个支持 Servlet2.3 或 JSP1.2 及以上版本的 JSP 容器或应用服务器。

如果运行的是最新版本的Sun应用服务器，那么已经包含了JSF和JSTL，不需要再做任何操作。如果运行的是Tomcat 5.0 或 5.5，则需要从 <http://java.sun.com/j2ee/javaxserverfaces/download.htm>下载JSF，还应从 <http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html>下载JSTL。然后，把它们解压缩到某个文件夹，并记住这个目录。

Tomcat 中运行的应用程序使用 JSF 和 JSTL 的方法有两种。这两种方法都包含了下面八个 JAR 文件，它们位于安装包的 lib 目录下，把这些 JAR 文件复制到可以被服务器或 Web 应用访问的目录中。

JSF 有六个 JAR 文件：commons-beanutils.jar，commons-collections.jar，commons-digester.jar，Commons-logging.jar，jsf-api.jar 和 jsf-impl.jar。JSTL 有两个 JAR 文件：jstl.jar 和 standard.jar。

某些情况下，如果只想让特定的应用访问 JSF 和 JSTL，那么可以把这两个 API 的 JAR 文件放到相应 Web 应用的 WEB-INF/lib 目录下，这样就只有这个应用有权限访问这些库文件。如果还有另外的 JSF 应用，那么这个应用需要访问它自己的 WEB-INF/lib 目录下的这些文件。

另外，如果有多个 JSF 应用，可以把这些 JAR 文件放到一个公共目录下。对于 Tomcat，这个目录是 tomcat_dir/common/lib。当把这些 JAR 文件放入这个公共目录之后，应用服务器里的每个应用都有权限访问它们。注意，如果当服务器在运行时把这些 JAR 文件复制到公共目录里，那么可能需要重启 Tomcat 服务器以便载入这些新的 JAR 文件。

4.5.2 JSP页面中使用JSF

前面简要介绍了 JSF 的概念，现在可以直接来创建和部署一个简单的 JSF 应用，这个应用将展示如何在 JSP 页面中用 JSF 的 MVC 实例：模拟航班订票系统。JSP 页面是容易实现和部署的，这个例子将清楚地说明如何在 Web 应用中使用 JSF。此外，JSF 规范要求所有的 JSF 实现必须支持 JSP 页面和提供与 JSF 及 UI 组件相对应的定制操作。

1. JSF实现包括的定制操作库

Sun 提供的 JSF 实现包括两个可以在 JSP 页面中使用的定制操作库：HTML 定制操作和核心定制操作。根据不同组件使用的 render 工具不同，HTML 定制操作用于创建各种 HTML 元素。如表 4-6 所示，HTML 定制操作分为 5 类或 5 个元素：输入（input）、输出（output）、选择（selection）、命令（command）和其他（miscellaneous）。

表 4-6 HTML 定制操作

分 类	元 素	目 的
输入(input)	h:inputHidden,h:inputSecret, h:inputText,h:inputTextarea	创建各种输入元素
输出(output)	h:message,h:messages, h:outputFormat, h:outputLabel, h:outputLink, h:outputText	创建各种输出元素
选择(selection)	h:selectBooleanCheckbox, h:selectManyListbox, h:selectManyMenu, h:selectOneListbox, h:select OneMenu, h:selcetOneRadio	创建下拉菜单、列表框(list h:selectManyCheckbox, box)、单选按钮(radio button)、复选框(checkbox)
命令(command)	h:commandButton, h:commandLink	创建表单提交的按钮或者链接
其他(miscellaneous)	h:dataTable,h:form, h:graphicImge, h:panelGrid, (panel)等	创建其他各种 HTML 元素，如表(table)、表单(form)和面板 h:panelGroup,h:column

由核心定制操作创建的 UI 元素不依赖于 render 工具。通常这些操作和表 4-6 中的 HTML 操作关联起来使用，从而可以改变 HTML 操作的行为。表 4-7 列出了所有的核心定制操作，分类如下：

表 4-7 核心定制操作

分 类	元 素	目 的
转换器(Converter)	f:convertDateTime, f:convertNumber, f:converter	标准的转换器

分 类	元 素	目 的
监听器(Listener)	f:actionListener, f:valueChangeListener	为组件设定一个监听器
其他(Miscellaneous)	f:attribute,f:loadBundle, f:param, f:verbatim	添加属性或参数、载入资源包(resource bundle) 逐字输出 HTML 模板文本
选择(Selection)	f:selectItem,f:selectItems	为 HTML 选择元素设定选择项
验证器(Validators)	f:validateDoubleRange, f:validateLength, f:validateLongRange	标准的验证器
视图(View)	f:facet,f:subview,f:view	创建 JSF 视图或子视图(subview)

技巧：如果将 JSF 实现解压缩到硬盘上，那么会产生一个定制操作的文档目录 jsf_root/tlddocs/index.html。这个文档类似于用 Java 源代码文件创建的 Javadoc 文档。如何具体使用定制操作可参考这个文档。

2. 创建一个简单的JSF应用

下面的例子是一个简单的模拟航班订票系统。

1) 实现一个 JavaBean

首先我们来看一下这个 Web 应用业务层的 JavaBean 类。这个 Bean 通过 JSF 系统与表示层相连接，它表示在系统中查询航班所必需的信息。如例 4-22 所示的 FlightSearch 类，用来保存用户输入的查询参数。只选择其中的一部分参数，如起飞机场、目的机场、起飞日期时间、到达日期时间，等等。

【例 4-22】FlightSerch.java。

```
package com.après.jsf;

public class FlightSearch {
    String origination;
    String destination;
    String departDate;
    String departTime;
    public String getDepartDate() {
        return departDate;
    }
    public void setDepartDate(String departDate) {
        this.departDate = departDate;
    }
    public String getDepartTime() {
        return departTime;
    }
    public void setDepartTime(string departTime) {
        this.departTime = departTime;
    }
    public String getDestination() {
        return destination;
    }
}
```

```

public void setDestination(string destination) {
    this.destination = destination;
}
public String getOrigination() {
    return origination;
}
public void setOrigination(string origination) {
    this.origination = origination;
}
}

```

观察这个类，会发现它是一个标准的 **JavaBean**。这个类没有明显的构造函数，所以编译器会给它一个默认的无参数的构造函数。该类的类变量用来保存参数，有读取和设置每个类变量的方法。也就是说，该类的所有属性对于整个 **Web** 应用都是可读写的。这样就允许该应用的某一部分可以设置这些属性，而应用的另一部分可以读取这些属性。

部署这个例子之前，需要把 **FilghtSearch.java** 源码编译成一个类文件。因为这个源文件只引用了 **java.lang** 包，且没有用任何其他 API 或类，所以不必重设类路径就应该可以编译。用你的 **IDE** 编译这个类，或者用命令行 **javac** 来编译。

2) 实现视图 (view) 组件

这个例子接下来的部分是接收用户输入航班查询条件的 **Web** 页面。这是一个 **JSP** 页面，它包含起点、终点、离港日期时间的输入域。例 4-23 所示是此应用的初始化页面 **searchForm.jsp**，该页面用了输入字段，但与以前的使用稍有不同。

【例 4-23】searchForm.jsp。

```

<html>
    <%@ taglib uri= "http://java.sun.com/jsf/core" prefix= "f" %>
    <%@ taglib uri= "http://java.sun.com/jsf/html" prefix= "h" %>
    <f:view>
    <head>
        <title>Freedom Airlines Online Flight Reservation System</title>
    </head>
    <body>
        <h:form>
            <h2>Search Flights</h2>
            <table>
                <tr><td colspan= "4">where and when do you want to travel?</td></tr>
                <tr>
                    <td colspan= "2">Leaving from:</td>
                    <td colspan= "2">Going to:</td>
                </tr>
                <tr>
                    <td colspan= "2">
                        <h:inputText value= "#{flight. Origination}" size= "35"/>
                    </td>
                    <td colspan= "2">
                        <h:inputText value= "#{flight.destination}" size= "35"/>
                    </td>
                </tr>
            </table>
        </h:form>
    </body>
</html>

```

```

        </td>
    </tr>
    <tr>
        <td colspan= "2">Departing:</td>
    </tr>
    <tr>
        <td>
            <h:inputText value= "#{flight.departDate}"/>
        </td>
        <td>
            <h:inputText value= "#{flight.departTime}"/>
        </td>
    </tr>
</table>
<p>
<h:commandButton value= "Search" action= "submit"/>
<./p>
</h:form>
</body>
</f:view>
</html>

```

前面章节讨论了如何从 JSP 页面中消除 Java 代码，JSF 也具有这样的能力。从例 4-23 看到这个页面中没有一行声明语句或脚本程序，其中只有两个 `taglib` 指令、一些标准的 HTML 标签和一些定制操作的标签。在 `taglib` 指令中定义页面所用到的标签库，并且定义的这些标签以 `f:` 或 `h:` 开头。

从 `taglib` 指令可以看出，使用前缀 `f:` 的标签为页面提供核心的 JSF 功能，而使用前缀 `h:` 的标签为页面提供 HTML 元素。页面中有一个 JSF 标签，即 `view` 标签，任何包含了 JSF 元素的页面必须用 `view` 标签作为最外层的 JSF 标签。该页面其余的 JSF 标签用来创建页面的 HTML 元素。其中 `form` 标签创建 HTML 表单，`input` 标签创建表单里的输入文本字段，而 `commandButton` 标签创建表单里的一个按钮。

如果熟悉 HTML 表单，那么应该知道每个 HTML 表单必须有一个 `action` 属性，另外还可能有一个 `method` 属性。其中 `action` 属性告诉 Web 浏览器往哪里提交表单数据，`method` 属性告诉浏览器究竟提交 GET 请求还是 POST 请求。而 JSF 标签不用这些属性，JSF 规范要求把所有的 JSF 表单从其存储位置发送到相同的 URL（如果应用把表单数据提交给同一个页面，那么如何处理数据和在不同页面间跳转？这个问题将在下面详细解答）。JSF 规范要求所有的 JSF 表单都用 POST 方法来提交表单数据给 Web 应用。因为这里的 `method` 和 `action` 的值是固定的，且程序员不能修改它们，所以也不需要再在 JSF 标签中设定它们。

还要注意的，`input` 标签的 `value` 属性的语法和标准的 HTML 语法不同。要理解表达式语言（EL），其中 EL 语法的 `#{}` 符号。JSP 页面用 EL 表达式（如 `#{flight.origination}`）来访问页面的对象属性。点符左边的名称是页面可访问的对象名；点符右边的名称是被访问对象的属性。在例 4-22 中可以看到一个属性名为 `origination`，相应地它还有 `set` 和 `get` 方法。当 JSP 页面访问这个 `JavaBean` 时，那么在页面中用 `#{flight.origination}` 表达式就可以读写

origination 属性。searchForm.jsp 页面结合输入域来使用这个表达式。当提交该页面时，在这些域中填入的值将被用来设置 JavaBean 的属性值。

对于实际的航班在线订票 Web 应用系统，用户提交请求之后系统查询和显示航班。但是在本例中，只是把查询参数简单返回给用户，这个功能用 searchResults.jsp 页面来实现。

【例 4-24】 searchResults.jsp。

```
<html>
  <%@ taglib uri= "http://java.sun.com/jsp/core" prefix= "f" %>
  <%@ tahlb uri= "http://java.sun.com/jsp/html" prefix= "h" %>
  <f:view>
    <head>
      <title>Freedom Airlines Online Flight Reservation System</title>
    </head>
    <body>
      <h3>You entered these search parameters</h3>
      <p>Origination: <h:outputText value= "#{flight.origination}"/>
      <p>Depart date: <h:outputText value= "#{flight.departDate}"/>
      <p>Depart time: <h:outputText value= "#{flight.departTime}"/>
      <p>Destination: <h:outputText value= "#{flight.destination}"/>
      <p>Trip type : <h:outputText value= "#{flight.tripType}"/>
    </body>
  </f:view>
</html>
```

该页面和 searchForm.jsp 页面一样，最外边的 JSF 标签是 f:view。在 view 标签里，页面用了 h:outputText 标签来向页面输出文本。outputText 标签和 inputText 标签一样，也用 #{object.property} 语法来访问页面对象的属性。这里的 JavaBean 对象名称是 flight。上面的 outputText 标签从对象中读取属性并且显示在 JSP 产生的 Web 页面中。

现在已经有了 Web 应用的三个主要部分：一个输入页面、一个输出页面和一个提供业务数据的 JavaBean。按照 MVC 模式，FlightSearch 是模型（model），而 searchForm.jsp 和 searchResults.jsp 是视图。但是目前为止还没有看到控制器，也没有解释控制器如何知道在哪里找到模型或视图，以及控制器如何知道 Web 应用的逻辑流程。在第 4.2.3 节的 MVC 例子里，需要在 JSP 页面中编写相应的控制流。而在目前的代码清单里，会发现 searchForm.jsp 或 searchResults.jsp 并没有包含任何页面到页面跳转的控制信息。下面看一下应该如何管理这些控制流程。

3) JSF 应用的控制流程

Web 应用的视图组件信息和控制流信息都包含在一个特定的配置文件 faces-config.xml 中，如例 4-25 所示。尽管 faces-config.xml 可以包含 Web 应用各种不同的信息，但是我们只需要做两件事：确定从 searchForm.jsp 到 searchResults.jsp 的控制流和确定该应用所使用的 JavaBean。

【例 4-25】 faces-config.xml。

```
<?xml version= "1.0"?>
<faces-config xmlns= "http://java.sun.com/xml/ns/javaee"
  xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance"
```



```

xsi:schemaLocation= "http://java.sun.com/xml/ns/javaee/web-
    facesconfig_1_2.xsd"
version= "1.2" >
<navigation-rule>
    <from-view-id>/searchForm.jsp</from-view-id>
    <navigation-case>
        <from-outcome>submit</from-outcome>
        <to-view-id>/searchResults.jsp</to-view-id>
        <redirect/>
    </navigation-case>
</navigation-rule>
<managed-bean>
    <managed-bean-name>flight</managed-bean-name>
<managed-bean-class>com.après.jsf.FlightSearch</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
</managed-bean>
</faces-config>

```

faces-config.xml 文件里通过 managed-bean 元素确定 Web 应用所有的 JavaBean。对于应用所用到的每个 JavaBean，都要有一个相应的 managed-bean 元素与之对应。例 4-25 中的 managed-bean 元素包含了下面三个子元素：第一个子元素是 JSP 页面所用 bean 的名称。例 4-25 里的名称是 flight，这就是为什么 searchForm.jsp 和 searchResults.jsp 可以用#{flight...} 表达式来访问 bean 的实例。第二个子元素是 JavaBean 类的完整类名。这个类名告诉 JSP 容器应该载入哪个类和创建哪个 JavaBean 实例。第三个子元素确定了对象的作用域。作用域为 session 说明对象存在于用户和应用的整个交互过程中。容器必须保存跨多个请求/响应周期的对象，直到用户会话结束为止。

另外，faces-config.xml 文件通过 navigation-rule 元素告诉控制器如何对应进行导航。本例只需要一个 navigation-rule 元素。通常，navigation-rule 元素要有一个起始页面、一个条件以及当条件满足时跳转到页面。

本例中的起始页面是 searchForm.jsp。如果用提交的结果作为页面的请求，那么将跳转到 searchResults.jsp 页面。从例 4-23 可以发现 commandButton 元素包含一个 submit 操作，当单击这个按钮时将提交该表单，这个操作与 navigation-rule 中的 from-outcome 相匹配。Navigation-rule 元素还包含着一个空的 redirect 元素。这个元素可以引发浏览器重定向到 searchResults.jsp 页面来进行响应，并且还会更新浏览器的地址栏。如果不用这个元素，那么响应会被正确创建和发送到浏览器，但是浏览器的地址栏不会被更新，仍然显示提交前页的地址。

这个 Web 应用最后还有一步。在前面的许多例子中，用户访问 Web 应用都要首先访问一个默认的页面。本例中的默认页面是一个标准的 HTML 页面 index.html，它将重定向至 JSF 应用的相应的 URL。

【例 4-26】index.html。

```

<html>
<head>
    <meta http-equiv= "Refresh" content= "0; URL=searchForm.faces"/>

```

```
</head>
```

```
</html>
```

从这段代码可以看到重定向的 URL 是 `searchForm.faces`。但是，我们的应用里并没有名为 `searchForm.faces` 的组件。那么 Web 应用怎么知道要用哪个页面呢？所有的 JSF 请求重定向到应用的控制器，这个控制器是一个 Servlet，它是 JSF 的部分参考实现。当部署这个例子的时候，设定了发送给控制器 Servlet 的 URL 应该是 `*.faces` 格式的。这个 Servlet 把 `searchForm.faces` 请求转换成 `searchForm.jsp`，然后再处理这个 JSP 页面，并向浏览器发送响应。

4) 在 Sun 应用服务器中部署此应用

正如前几章所述，可以用部署工具把应用部署到 Sun 的应用服务器。要部署第一个 JSF 例子，按照下面的步骤进行：

- ① 用工具创建一个 Web 应用。
- ② 向该应用添加 `index.html`、`searchForm.jsp`、`searchResults.jsp`、`faces-config.xml` 和 `FlightSearch.class` 文件，保证它们位于正确的位置。
- ③ 用名字 `Faces Servlet` 配置 Servlet 控制器 `javax.faces.WebappFacesServlet`。
- ④ 配置欢迎文件列表，其中用 `index.html` 作为欢迎页面。
- ⑤ 创建 Servlet 映射，让包含 `*.faces` 匹配符的 URL 可以映射到名为“Faces Servlet”的 Servlet。
- ⑥ 保存并把该应用部署到服务器。

5) Tomcat 服务器中部署此应用

要把这个应用部署到 Tomcat 服务器，需要编写一个 `web.xml` 部署描述文件，如例 4-27 所示。

【例 4-27】Jsf_Ex01 的 `web.xml`。

```
<?xml version= "1.0"?>
<web-app xmlns= "http://java.sun.com/xml/ns/javaee"
  xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation= "http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    version= "2.5">
  <display-name>Jsf_Ex01 - Simple JSF Application</display-name>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>java.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.faces</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

这个部署描述文件确定了应用的控制器 `Servlet(Faces Servlet)`，指定发送到控制器

Servlet 的请求的 Servlet 映射，并且设定该应用的欢迎文件。

正如本节开头一节“安装 JSF”所描述的，需要把 JSF 和 JSTL 的 JAR 文件复制到应用的 WEB-INF/lib 目录下，或者复制到 Tomcat 的 common/lib 目录下。然后把整个目录复制到 Tomcat 的 Webapps 目录里，这样部署就算完成了。

或者，在创建完目录结构之后，用 Java 的 jar 命令把所有的应用文件打包到一个 WAR 文件中。创建完这个 WAR 文件，把它复制到 Tomcat 的 webapps 目录里。Tomcat 将自动解压缩这个 WAR 文件并运行该应用。

6) 运行这个应用

当把这个应用成功地部署到应用服务器之后，打开一个 Web 浏览器输入 `http://localhost:8080/Jsf_Ex01`（和过去一样，应把这里的 localhost 和 8080 换成你自己的值；如果部署的 JSF 应用使用的是其他的环境名，那么应该把 Jsf_Ex01 换成相应的应用环境名）。

应用一开始首先调用 `index.html` 页面，然后立即重定向到 `searchForm.faces`。根据 `web.xml` 中的 Servlet 映射，这个请求会被转到 Faces Servlet 控制器。该控制器知道与 `searchForm.faces` 请求相对应的是 `searchForm.jsp` 组件。如果这是第一次访问 `searchForm.jsp` 页面，那么服务器将对它进行编译，然后再发送给浏览器。

如果查看这个页面的源文件，会发现 JSF 的 form 标签已经被转换为 HTML 的 form 标签。正如先前所提到的，这个表单的方法（method）是 POST，它的操作（action）是 `searchForm.jsp` 页面的 URL。虽然看起来这个表单提交给了同一个 JSP 页面，但是在 `faces-config.xml` 文件中控制器 Servlet 会用导航规则来保证页面导航的正确性。另外，每个 JSF 输入标签已经被翻译为 HTML 的输入（input）标签。

在每个输入域中输入值。由于 FlightSearch JavaBean 每个属性都是 String 类型的，所以每个文本域都可以输入任意值。完成输入后，单击查询（Search）按钮。这个请求会被传递给服务器，然后由 `searchResults.jsp` 产生响应并发送回浏览器。

习 题 4

1. 配置并使用 JSP 的开发、应用环境。
2. 编写一个对用户提交的表单内容在浏览器中回显出来的程序。
3. 编写一个对带有单选按钮的表单进行处理的程序。
4. 编写一个计算 $1+2+3+\cdots+N$ 的 JavaBean。
5. 编写一个使用 Cookie 记录浏览者信息的程序。
6. 编写一个对已经建立的学生成绩表进行查询，删除操作的 JSP 程序。
7. 什么是 JSTL？如何在 JSP 中使用 JSTL？
8. 什么是 JSF？如何在 JSP 中使用 JSF？

第 5 章 Java EE 轻型框架技术

本章主要介绍 Java EE 的轻型框架 Hibernate、Struts2、Spring 的基础、实例、配置文件参数、用 MyEclipse 支持创建 Java EE 轻型框架、MVC 结构（当然也使用了其他平台创建，如 Eclipse，见 5.3.3）等。最后还给出开发 Struts2、Hibernate、Spring 集成程序的实例，以便读者对 Java EE 轻型框架技术有一个基本的了解，在开发中小型企业级项目时能游刃有余。

5.1 Java EE 轻型框架技术概述

EJB 是 Java EE 技术框架中的精华，它主要针对大型企业的软件项目，使用了 EJB 的 Java EE 框架适合于大型企业或大型项目等。对于大多数中小型企业或中小型项目，反而需要敏捷轻型的框架，不需要或暂时不需要分布式的设计模式，这样既可以加速企业管理项目的开发速度，同时节约了开发成本，而且也满足了企业的要求。目前这种轻型框架是非常流行的，如 Hibernate、Struts、Spring、WebWork、Tapestry、JSF 等。

一般认为，EJB 框架为重型框架（见第 6 章），因为它需要启动应用服务器加载 EJB 组件。软件架构复杂，启动加载时间长，需要的系统昂贵，软件费用花销也很大。而轻型框架则不同，它不需要昂贵的设备和软件费用，系统搭建容易，服务器启动快捷，非常适合于中小型企业或项目。

5.1.1 轻型框架的流行

框架，即 Framework，其实就是某种应用程序的半成品，把不同应用程序中有共性的一些内容抽取出来，做成一个半成品程序，这样的半成品就是所谓的程序框架。

1. 为什么要使用框架

软件系统发展到今天已经很复杂了，特别是服务器端软件，涉及的知识、内容、问题非常多。在某些方面使用成熟的框架，就相当于基础工作已经完成，只需要集中精力完成系统的业务逻辑设计即可。这样每次开发就不用从头做起，而是在这个基础上开始搭建。

使用框架的最大好处不仅在于减少重复开发工作量、缩短开发时间、降低开发成本。同时还有其他优势，例如，使程序设计更合理、程序运行更稳定，等等。基于这些原因，目前的软件开发都会选用某些合适的开发框架，从而达到快捷、高效的目的。

2. 如何选择框架

选择合适的框架是一个谨慎的事情。框架选择得好，系统开发轻松、代码量少，运行稳定。反之，则系统结构混乱、不易维护和调试。选择框架可以参照以下原则：

（1）框架的学习一定要简单，上手一定要快。

（2）一定要能得到很好的技术支持。在应用的过程中，或多或少都会出现这样或者那样的问题，如果不能很快地解决，会对整个项目开发带来影响。

(3) 开发框架结合其他技术的能力一定要强。例如，在逻辑层要使用 Spring 或者 EJB 3.0，那么该开发框架一定要能很容易、很方便地与之结合。

(4) Web 开发框架的扩展能力一定要强。任何框架都有力所不及的地方，这就要求该框架有很好的 Web 开发框架的功能，以满足新的业务需要，同时要注意扩展的简单性。

(5) Web 开发框架最好能提供可视化的开发和配置，可视化开发对开发效率的提高，已经得到业界公认。

(6) 开发框架的设计结构一定要合理，应用程序会基于这个框架构建，框架设计得不合理会大大影响整个应用的可扩展性。

(7) 开发框架运行要稳定，运行效率高。框架的稳定性和运行效率直接影响到整个系统的稳定性和效率。

(8) 任何开发框架都不可能是十全十美的，也不可能适应所有的应用场景。也就是说，任何开发框架都有它适用的范围，因此选择的时候要注意判断应用的场景和开发框架的适用性。

5.1.2 流行的轻型框架组合

在目前流行的框架中，开发者几乎可以根据自己的喜好进行任意组合，在实际工程中也取得了成功。下面列出一些常见组合，且这些组合会随着新技术的出现而发生变化。

- (1) JSP + Servlet + JavaBean + JDBC。
- (2) Struts + MySQL + JDBC。
- (3) Hibernate + JDBC + JSP。
- (4) Struts + Hibernate。
- (5) Hibernate + Spring。
- (6) Spring + Struts + JDBC。
- (7) Struts + Hibernate + Spring。
- (8) Struts + EJB。
- (9) JSF + Hibernate。
- (10) Typetry + Hibernate + Spring。
- (11) Freemake + Struts + Hibernate + Spring。
- (12) JSP + EJB + Oracle。

由于篇幅所限，以上仅列出一些常见的组合。

目前 Java EE 轻型框架发展非常迅速，目前非常流行的有以下几种：Hibernate 框架（对 JDBC 进行了轻量级的对象封装）、Struts 框架（MVC 模式的典型实现方式）、Spring 框架（对 Bean 的生命周期管理、全方位的应用程序框架），等等。

5.1.3 轻型框架的MyEclipse环境

1. MyEclipse简介

下面对集成了轻型框架等众多插件的 MyEclipse 开发环境做简单的介绍。

来自 Genuitec 的 MyEclipse 是 Eclipse 的一个插件，也是一个功能强大的 Java EE 集成开发环境，支持完备的应用程序开发周期，包括：编写程序代码、测试程序并进行调试、应用程序部署以及效能调整等阶段。MyEclipse 为 Eclipse 提供了一个大量私有和开源的 Java

工具的集合，解决了各种开源工具不协调的缺点。MyEclipse 包含生成 Struts Web 应用的自动工具，还包含其他的工具，如配置 Hibernate 框架的数据库连接及 SQL 浏览器，同时还整合了 Spring 框架。

MyEclipse 的实际价值来自包含的发布包中的大量工具，如 CCS、JS、HTML 和 XML 的编辑器，帮助创建 EJB 和 Struts 项目的向导，并产生项目的所有主要的组件，如 action、session bean、form 等。

2. MyEclipse环境的搭建

(1) MyEclipse 的下载。MyEclipse 是开源的，在 <http://www.myeclipseide.com/>中就可以下载到试用版，如 MyEclipse_6.0GA_E3.3_FullStackInstaller。

(2) MyEclipse 的安装。MyEclipse 的安装选择默认选项即可。

(3) MyEclipse 配置。

① 配置 JDK、JRE：JDK 是 Java 的开发环境，JRE 是 Java 的运行环境。因此，编写 Java 程序的时候需要 JDK，而运行 Java 程序的时候需要 JRE。在 JDK 里面已经包含了 JRE，因此也可以正常运行 Java 程序。在 MyEclipse 中内嵌了 Java 编译器。在这里指定自己安装的 JDK。

② 配置 MyEclipse 与 Tomcat 集成：启动 MyEclipse，选择菜单 Windows→Preferences 命令，显示 MyEclipse 配置对话框，单击左边目录树中的 MyEclipse→Application Servers→Tomcat→Tomcat 6.x 选项，并激活 Tomcat 6.x，设置 Tomcat 的路径。选择对话框左边目录树中的 MyEclipse→Application Servers→Tomcat→Tomcat 6.x→JDK，将 Tomcat 6.x 下的 JDK 设置为前面设置的名为 jdk 的 Installed JRE。测试是否配置成功：在 MyEclipse 的工具栏单击“start→stop/restart MyEclipse Server”按钮，选择 Tomcat 6.x→Start，在控制台会出现启动 Tomcat 的信息。打开浏览器，输入 <http://localhost:8080/>，如果配置成功，则出现 Tomcat 首页。

5.2 Struts2 框架

本节介绍 MVC 模式的典型实现方式——Struts 开源框架。由于篇幅有限，只讲解 Struts2 框架的主要内容。

5.2.1 Struts框架及其MVC结构

Struts 是建立在 JSP、Servlet 和 XML 等相关开发技术基础之上的一种主流的、经典的 MVC 设计模式框架。

Struts 是 Apache 基金会的一个开源项目，其网址是 <http://struts.apache.org/>。作为一个通用的设计框架，Struts 可以让开发人员把主要精力集中在如何解决实际业务处理问题上，同时，Struts 框架也允许开发人员根据实际需要进行扩展和定制，从而能更好地适应用户的需求。使用 Struts 框架可以很好地实现代码重用，快速开发具有强大可扩展性的 Web 应用。

Apache 的 Struts 项目提供了 Struts 框架的两个主要版本：Struts1 和 Struts2。Struts1 被认为是广泛的、用于开发 Web 程序的 Java 开源框架，是解决很多软件设计问题的上佳选择，目前其稳定版本是 Struts1.3.8。Struts2 来源于另一个开源设计框架 WebWork2（下载网址 <http://www.opensymphony.com/webwork>），后来被并入 Apache 的 Struts 项目，逐渐演化为 Struts2。它被认为是一种解决困难问题的灵活方案，代表了应用趋势。

Struts 作为 MVC 模式的一种典型实现，对控制器(Controller)、模型 (Model) 和视图 (View) 都提供了现成的实现组件，其实现方式如图 5-1 所示。

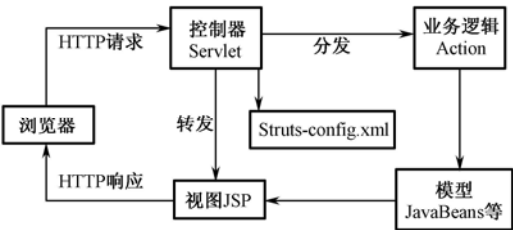


图 5-1 Struts MVC 结构图

1. 控制器 (Controller) 部分

Struts 中的控制器部分是通过 Servlet 实现的，是 Struts API 中 ActionServlet 类的实例。该类继承了 javax.servlet.http.HttpServlet，其作用是接收客户端浏览器的请求，然后选择执行相应的业务逻辑，再把结果送回客户端。

Action 对象是开发者定义的类，必须遵循 Struts 的规范，开发者可以实现具体的业务逻辑或者调用业务逻辑模块。Action 对象在进行了业务逻辑处理后，会将应用程序流程转到合适的视图组件，最后将响应送到浏览器。Struts 提倡 Action 只用来决定“做什么”，可以将其归纳为 Controller 的组成部分。

此外，Struts 用 ActionForm 对象自动接收客户端表单数据，可以看做模型和视图的中介，一般用于存储从视图中获取的数据，并提供给其他模型或视图使用，有时候也可以把 ActionForm 看做模型的一部分。

2. 模型 (Model) 部分

Model 从概念上可以分为两类：系统的内部状态和改变系统状态的动作，一般由 JavaBean 或者 EJB 等组成，也可以由其他开源框架组成。Struts 架构中使用 JavaBean 来提供具体的业务逻辑，即“怎么做”。对于复杂的系统可以使用 EJB 等组件或者其他开源框架来实现系统状态的维护，有时候也可以把 ActionForm 看做模型的一部分。

3. 视图 (View) 部分

视图部分可以用 JSP 来实现，Struts 还提供了丰富的自定义标签库增强显示功能，减少或者避免在 JSP 中使用 Java 代码。

5.2.2 Struts2 与WebWork在代码重用性上的优势

Web MVC 框架技术在 Java Web 开发领域得到了广泛应用，但大部分框架存在开发难度高、代码编写量大、功能模块难以测试等问题。WebWork 框架的出现很好地解决了这些问题。本节主要分析了 WebWork 框架的工作原理、架构构成和组件实现机制，最后给出了一个基于 WebWork 框架的 Web 应用开发及运行环境实例。

1. 引言

为适应 Web 应用系统规模不断扩大，复杂性急剧增加的发展趋势，Web 开发领域普遍采用三层体系结构与面向对象技术相结合的软件开发模式。在 Java Web 领域涌现出了众多基于 MVC (Model View Controller) 的设计框架，如 Struts, JSF, Tapestry 等。其中，Struts 凭借其有力的技术支持得到了广泛应用，但是，由于 Struts 是基于 JSP/Servlet 架构发展起来

的，其业务逻辑控制器内充满了大量的 Servlet API，使得功能模块难以测试，并且由于其代码严重依赖于 Struts API，属于侵入式设计，一旦系统需要重构时，导致代码重用性降低。因此，Struts 大大增加了前期开发的难度和开发成本。

为解决这一系列问题，OpenSymphony 组织开发出了 WebWork 开源架构。WebWork 能够提高开发者的生产效率，简化代码的编写与维护，加强组件开发并提高代码的重用性。

本节主要介绍 WebWork 的架构、工作原理及组件运行机制。通过对 WebWork 从最初 URL 请求到最终用户视图展现这一 MVC 过程的阐述，深入分析 WebWork 的初始化工作和 Interceptor、Action 等核心组件运行机制。

2. WebWork的工作原理

当前的 WebWork 框架由 XWork 和 WebWork2 两个项目组成，如图 5-2 所示。

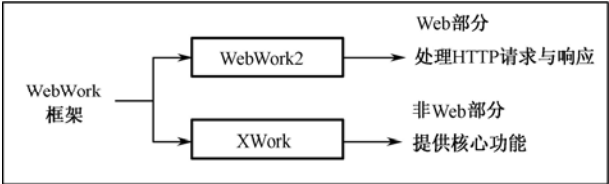


图 5-2 WebWork 框架构成

其中，XWork 是一个标准的命令模式框架，与 Web 层完全脱离。WebWork2 建立在 XWork 之上，处理 HTTP 的请求和响应。

WebWork 的官方网站上提供了一个完整的 WebWork 架构图。它描述了从客户端的一次请求到最后服务器端响应的具体执行过程，如图 5-3 所示。

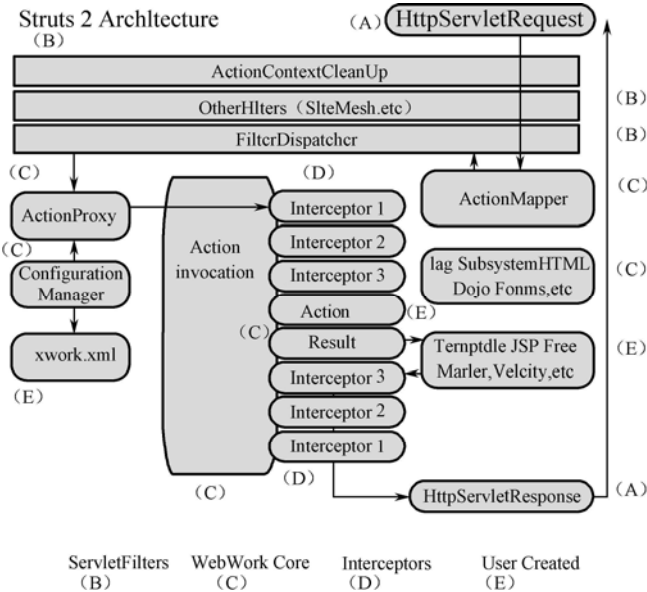


图 5-3 WebWork 架构图

图 5-3 中，A、B、C、D、E 五个部分表示的含义介绍如下。

A：表示客户端的一次 HTTP 请求，以及服务器端处理结束之后的一次响应。

B：表示一次 HTTP 请求所要经过的 Filter 过滤器。图中最后一个过滤器 FilterDispatcher 是 WebWork 的前端控制器，也是 WebWork 的核心控制器。

- C: WebWork 框架的核心部分，描述了 Action 被执行前后的组建运行机制。
- D: 表示拦截器。当 WebWork 截获 Action 请求时，在 Action 执行之前或之后调用拦截器方法。通过这样的方式，WebWork 实现了用插拔的方式将功能注入到 Action 中。
- E: 表示使用 WebWork 进行 Web 应用开发时，需要开发的程序模块，包括：Action 类，页面模板和配置文件 xwork.xml 等。

3. WebWork的关键技术

由于 WebWork 框架是一个重量级的 MVC 框架，涉及的内容较多，本文主要针对 WebWork 的前端控制器，以及与 Action 获取和处理相关的重要技术进行研究。

1) WebWork 的前端控制器

WebWork 的前端控制器 FilterDispatcher 是 WebWork 框架的核心控制器，它是 WebWork 框架的开始，运行在 Web 应用的整个生命周期中。FilterDispatcher 通过调用 Configuration 类完成 WebWork 环境的初始化；然后，负责拦截所有的 HTTP URL 请求，当请求符合过滤要求时，将请求转入到 WebWork 框架，并调用 ActionMapper 将请求封装成为符合 WebWork 处理要求的 ActionMapping；最后，通过 XWork 的 ActionProxy 完成对该 ActionMapping 的相应操作，从而达到处理 HTTP 请求的目的。

其中，WebWork 环境的初始化工作较为直观，本节重点介绍 FilterDispatcher 对 URL 的过滤与封装。

当一次 HTTP 请求到达 WebWork 的前端控制器时，FilterDispatcher 首先会根据请求的 URL 解析出对应的 Action 名，然后咨询 ActionMapper 该 Action 是否可以被执行。如果可以被执行，前端控制器就把工作委派给 ActionProxy，并咨询 XWork 的配置管理器，读取在 xwork.xml 文件中定义的配置信息。具体实现流程如图 5-4 所示。

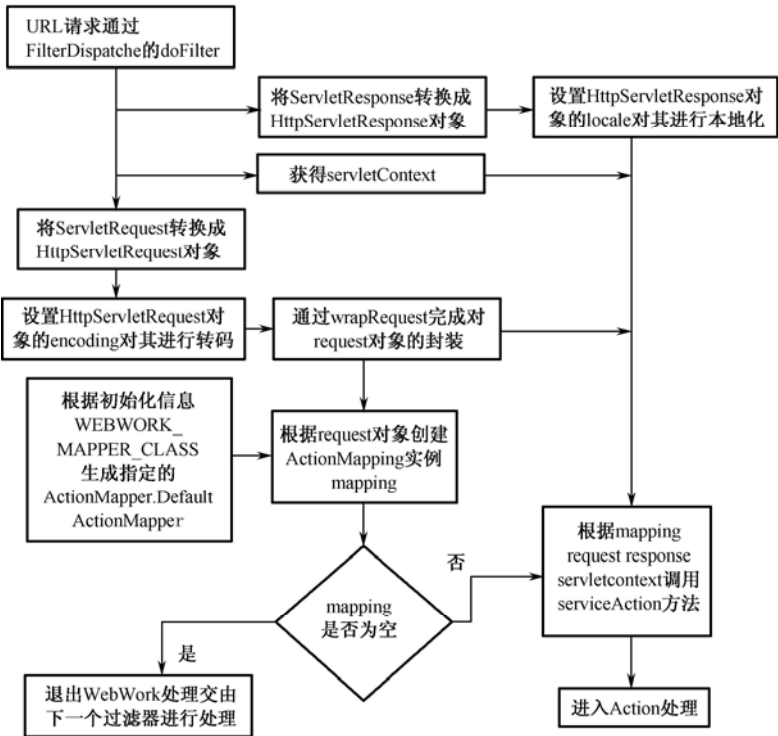


图 5-4 FilterDispatcher 对请求的过滤与封装

其间，FilterDispatcher 就像 WebWork 与 XWork 之间的桥梁，完成了 HTTP 请求与相应命令模式动作、结果的转换，同时 FilterDispatcher 也是连接 WebWork 各个部分的主线。

2) 通过 Interceptor 使用通用的功能和特性

从 WebWork 的架构图中可以清楚地看到，WebWork 的 Action 是开发人员使用 WebWork 进行编程的核心部分，它反映了 Web 应用的功能需求。

Interceptor 是 WebWork 的拦截器，WebWork 在执行 Action 之前或之后调用 Interceptor 方法。具体来说，Interceptor 在某个事件发生之前进行拦截，并插入某些相应的处理过程。这个过程类似于 Servlet 2.3 规范中引入的 Filter 过滤器，但是 XWork 的 Interceptor 与 Servlet 没有任何关系，因此，Interceptor 将很多通用的功能从 Action 中独立出来，大量减少了 Action 的代码，增强了代码的重用性和灵活性。Action 执行前后对 Interceptor 的调用如图 5-5 所示。

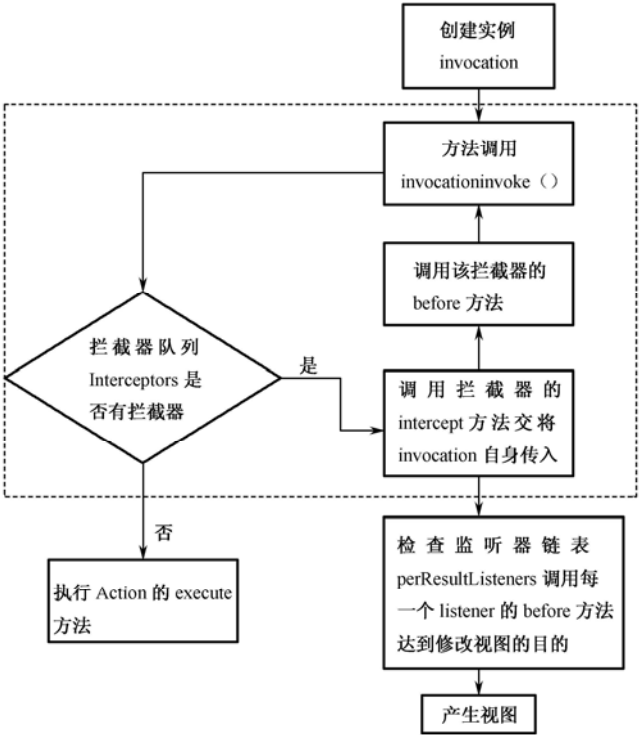


图 5-5 Action 基本处理流程

在执行 Action 的 execute 方法之前，拦截器队列中的 Interceptor 按开发人员配置好的顺序依次递归调用，直到最后一个 Interceptor 的 before 方法被执行完并返回结果 result 后，在 invoke 方法中执行表 5-1 所列的语句，实现对 Action 的 execute 方法的调用。

表 5-1 DefaultActionInvocation 调用 action 的实现

```
resultCode = invokeAction(getAction().proxy.getConfig());
```

之后再依次执行各 Interceptor 对象的 after 方法直到结束。由此，before 方法在 Action 执行前调用，after 方法在 Action 执行之后运行，实现了以插拔的方式将通用功能注入到 Action 中。开发人员可以在配置文件中组装自己的 Action 用到的 Interceptor。一旦行为需求发生变化，不必修改很多类，只要修改该行为配置即可。

WebWork 框架的很多功能都是以拦截器的形式提供的，例如，参数组装、验证、国际化、文件上传，等等，所以 **Interceptor** 是 **WebWork** 的核心内容之一。

3) 通过 **Action** 完成模型的处理

对于 **Web** 应用开发而言，与 **Web** 容器的交互大多集中在 **Session** 和 **Parameter**，这使得表现层与逻辑层没有完全解耦。既然 **Action** 是逻辑处理的入口，那么在编写 **Action** 的时候，也要面对同样的问题。对此，**WebWork** 实现了表现层与逻辑层的解耦，而 **Action** 的上下文是这个解耦过程的关键点。

Action 的上下文可以看做一个容器，它存放的是 **Action** 在执行时需要用到的对象，从实现的机制上看，**WebWork** 存在两种 **Action** 上下文：与容器无关的 **ActionContext** 和与容器相关的 **ServletActionContext**。

(1) 与容器无关的 **ActionContext**：**XWork** 在每次执行 **Action** 之前创建新的 **ActionContext**，**ActionContext** 为 **Action** 提供了与容器交互的途径，使 **Action** 不依赖于任何 **Web** 容器，不用和 **JavaServlet** 复杂的请求 (**Request**)、响应 (**Response**) 关联在一起。对请求 (**Request**) 的参数 (**Param**)，可以使用拦截器框架自动调用 **get()**和 **set()**方法设置到对应的 **Action** 字段中。例如，将 **HttpSession** 对象重新包装成一个 **Map** 对象，供 **Action** 使用，使得 **Action** 不用直接和底层的 **HttpSession** 打交道，从而实现 **Action** 与 **Web** 层解耦，保证 **XWork** 框架中的逻辑层与表现层无关，增加了 **Action** 代码的重用性。

(2) 与容器相关的 **ServletActionContext**：**ActionContext** 为 **Action** 提供了与容器交互的途径。使 **Action** 不用依赖于任何 **Web** 容器，但是对 **JavaServlet** 相关对象直接操纵却无能为力。在一些特殊的 **Web** 应用程序中也需要在 **Action** 里直接获取请求 (**HttpRequest**) 或会话 (**HttpSession**) 的一些信息，甚至需要直接对 **JavaServlet Http** 的请求(**HttpServletRequest**)和响应 (**HttpServletResponse**) 执行操作。因此，**WebWork** 针对该需求提供了与容器相关的 **Action** 上下文 **ServletActionContext**。

ServletActionContext 继承了 **ActionContext** 提供的所有功能，并提供了直接与 **JavaServlet** 相关对象访问的功能。**ServletActionContext** 的功能比 **ActionContext** 强大，但它是与 **Servlet API** 紧密耦合的，因此在 **WebWork** 框架中更多采用 **ActionContext**，在实现功能的同时，也使单元测试和在不同平台之间移植变得容易。

另外，在 **Action** 调用业务逻辑层处理事务的过程中，**WebWork** 集成了现有框架 **Spring** 作为自己的 **IoC** (**Inversion of Control**, 控制反转) 管理容器，将业务逻辑层 (**service**) 通过 **IoC** 的形式添加到 **Action** 中，使得控制层和逻辑层实现了解耦。

4) 基于 **WebWork** 应用的开发环境

基于对 **WebWork** 框架的 **Web** 应用程序，例如，**UserManage** 的设计与实现，可使用如表 5-2 所示的开发资源。

表 5-2 开发资源列表

资源名称	说 明
JDK 1.5	Java 开发工具包
Hibernate 3.0	MyApp 实例对象持久层容器
Oracle 9i	MyApp 实例使用的关系数据库
Tomcat	MyApp 实例使用的 Java Web 服务器
Eclipse 3.3.0	Java 应用开发 IDE 平台

资源名称	说 明
MyEclipse 6.0.1	Eclipse 的插件包，为 Eclipse 提供了一个大量私有和开源的 Java 工具的集合
WebWork 2.2.4	MyApp 实例应用 Web MVC 框架资源
Spring 2.0.1	MyApp 实例使用 IoC 容器

其中，很多资源是开源的，可以从网上免费得到。

5.2.3 Struts2 的引例、Filter及配置

1. Struts2 的引例（开发第一个Struts2 程序）

Apache Struts2 是一个为企业级应用打造的优秀的、可扩展的 Web 框架。该框架旨在充分精简应用程序的开发周期，从而减少创建、发布、应用所花费的时间。

在对 Struts2 做详细介绍之前，先感受一下 Struts2 的引例，开发一个用户登录 Web Project，如图 5-6 所示。如果输入姓名和密码是“scott”和“tiger”，则进入欢迎页面；反之，进入失败页面。

项目最后的目录树如图 5-7 所示。



图 5-6 用户登录界面

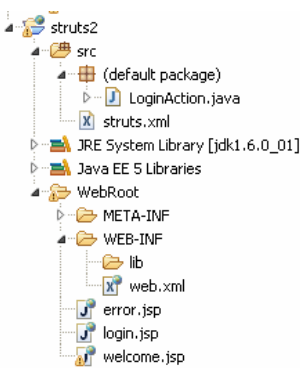


图 5-7 项目目录树

开发的步骤如下：

- (1) 下载 Struts2 框架；
- (2) 创建 Web Project；
- (3) 加载 Struts2 包；
- (4) 修改 web.xml；
- (5) 创建 login.jsp；
- (6) 实现控制器；
- (7) 配置 struts.xml；
- (8) 创建登录成功、失败页面；
- (9) 部署。

【例 5-1】Struts2 的引例（开发一个用户登录界面）。

(1) 下载 Struts2 框架。MyEclipse 6 没有对 Struts2 的支持，所以需要自己下载 Struts2 开发包。登录网页 <http://struts.apache.org/>，下载 Struts2 完整版，本书使用的是 Struts 2.0.11。将下载的 Zip 文件解压缩，它是一个典型的 Web 结构，该文件夹包含的文件结构如图 5-8 所示。



图 5-8 Struts 2.0.11
文件结构

apps: 包含基于 Struts2 的示例应用，对学习来说是非常有用的资料。
docs: 包含 Struts2 的相关文档，如 Struts2 的快速入门，Struts2 的文档、API 文档等内容。
j4: 包含让 Struts2 支持 JDK 1.4 的 JAR 文件。
lib: 包含 Struts2 框架的核心类库，以及 Struts2 的第三方插件类库。
src: 包含 Struts2 框架的全部源代码。

(2) 打开 MyEclipse，创建 Web Project，命名项目的名称为 Struts2。

(3) 加载 Struts2 包。将下载的 Struts2 包解压后得到 lib 文件夹下的

Struts2-core-2.0.11.jar、xwork-2.0.1.jar、ognl-2.6.11.jar、common-logging-1.0.4.jar 和 freemarker-2.3.8.jar 这五个必需的 jar 包，将它们复制到 Struts2/WebRoot/WEB-INF/lib 路径下。大部分时候，使用 Struts2 的 Web 应用并不需要用到 Struts2 的全部特性。

在目录树中，选中刚刚创建的 Struts2 工程，单击鼠标右键，选择 Build Path→Configure Build Path…，弹出对话框，单击“Add External JARs”按钮，将下载的五 jar 包添加到项目中。主要类描述如下：

common-logging.jar: 用于能够插入任何其他的日志系统。

ognl.jar: OGNL 表达式语言。

struts2-core.jar: Struts2 框架类库。

xwork.jar: XWork 项目，Struts2 就是在此基础上构建的。

freemarker.jar: 所有的 UI 标记模板。

(4) 修改 web.xml。打开 Struts2/WebRoot/WEB-INF/web.xml，修改成如下代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>
            org.apache.struts2.dispatcher.FilterDispatcher
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

(5) 创建 login.jsp。从用户请求开始，在 WebRoot 下创建一个简单的表单提交页面 login.jsp，方法是：单击 webRoot→new→jsp，在 File Name 中输入文件名“login.jsp”。

login.jsp 代码如下：

```
<%@page language="java" pageEncoding="gb2312"%>
<html>
    <head><title>登录页面</title></head>
```

```

<body>
    <form action= "login.action"method= "post">
        用户登录<br>
        姓名: <input type= "text"name= "username"/><br>
        密码: <input type= "text"name= "password"/><br>
        <input type= "submit"value= "登录"/>
    </form>
</body>
</html>

```

当表单提交给 login.action 时, Struts2 的 FilterDispatcher 将自动起作用, 将用户请求转发到对应的 Struts2 Action。

(6) 实现控制器。在 Struts2/src 目录下创建一个新类 LoginAction.java, 代码如下:

```

import com.opensymphony.xwork2.ActionSupport;
public class LoginAction extends ActionSupport{
    private String username;
    private String password;
    //处理用户请求的 execute 方法
    public String execute() throws Exception{
        //如果用户名为 "scott", 密码为 "tiger", 则返回成功 "success", 否则返回 "error"
        if(getUsername().equals("scott")&&getPassword().equals("tiger")){
            return "success";
        }
        else{
            return "error";
        }
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
    }
}

```

上面的 Action 类是一个普通的 Java 类。该类定义了两个属性: username 和 password。类变量的命名必须与在 login.jsp 中使用的文本输入框的命名严格匹配。在 Struts2 中, 类变量总是在调用 execute()方法之前被设置 (通过 setUsername(), setPassword()方法), 这意味着在 execute()方法中可以使用这些类变量, 因为在 execute 方法执行之前, 它们已经被赋予了正确的值。

(7) 配置 struts.xml。在 src 下生成文件 struts.xml (注意文件位置和大小写), 里面代码如下:

```

<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <package name= "struts" extends= "struts-default">
        <action name= "login" class= "LoginAction">
            <result name= "error">/error.jsp</result>
            <result name= "success">/welcome.jsp</result>
        </action>
    </package>
</struts>

```

映射文件定义了名为 login 的 Action。即当 Action 负责处理 login.actionURI 的客户端请求时，该 Action 将调用自身的 execute 方法处理用户请求，如果 execute 方法返回 success 字符串，请求被转发到/welcome.jsp 页面，如果 execute 方法返回 error 字符，则请求被转发到/error.jsp 页面。

(8) 创建登录成功、失败页面。经过上面的步骤，这个 Struts2 应用基本上可以运行了，但还需要为该 Web 应用增加两个 JSP 文件，两个 JSP 文件分别是 error.jsp 页面和 welcome.jsp 页面，将这两个 jsp 页面文件放在 Web-Root 下。

welcome.jsp 的代码如下：

```

<%@page language= "java" pageEncoding= "gb2312"%>
    <%@taglib prefix="s" uri="/struts-tags"%>
<html>
    <head><title>成功页面</title></head>
    <body>
        <s:property value="username"/>已经成功登录!
    </body>
</html>

```

代码中第二行标签库定义将前缀 s 和 uri 之间建立映射关系。前缀 s 指明了所有 Struts2 标签在使用的时候以“s:”开头。

<s:property value= "username">是一个使用自定义 property 标签的 JSP 页面。这个 property 标签包含一个 value 属性值，通过设置 value 的值，标签可以从 Action 中获得相应表达式的内容，这是通过在 Action 中创建一个名为 getUsername()的方法得来的。将以上代码保存在 welcome.jsp 文件中。

error.jsp 的代码如下：

```

<%@page language=" java" pageEncoding="gb2312"%>
<html>
    <head><title>失败页面</title></head>
    <body>
        登录失败!
    </body>
</html>

```

(9) 部署。选择工具栏中的按钮“Deploy MyEclipse J2EE Project to Server...”，将新建的 Web 项目部署到服务器 Tomcat 中。

选择项目为 Struts2，单击“Add”按钮，选择 Tomacat 6.x 作为服务器，单击“OK”

按钮。

启动 Tomcat，运行程序，在浏览器输入：`http://localhost:8080/struts2/login.jsp`。

2. Struts2 的 Filter（过滤器）

在项目的 `web.xml` 中，有一个名词 `Filter`，这是什么呢？它起了什么作用呢？

【例 5-2】 项目的 `web.xml` 中的 `Filter`。

```
...
<filter>
  <filter-name>struts2</filter-name>
  <filter-class>
    org.apache.struts2.dispatcher.FilterDispatcher
  </filter-class>
</filter>
<filter-mapping>
  <filter-name>struts2</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
...
```

`Filter` 是 Java 中常用的一项技术，过滤器是用户请求和处理程序之间的一层处理程序。这层程序可以对用户请求和处理程序响应的内容进行处理。过滤器可以用于权限控制、编码转换等场合。

`Servlet` 过滤器是在 Java `Servlet` 规范中定义的，它能够对过滤器关联的 URL 请求和响应进行检查和修改。`Servlet` 过滤器能够在 `Servlet` 被调用之前检查 `Request` 对象，修改 `Request Header` 和 `Request` 内容；在 `Servlet` 被调用之后检查 `Response` 对象，修改 `Response Header` 和 `Response` 内容。`Servlet` 过滤器过滤的 URL 资源可以是 `Servlet`、`JSP`、`HTML` 文件，或者是整个路径下的任何资源。多个过滤器可以构成一个过滤器链，当请求过滤器关联的 URL 的时候，过滤器链上的过滤器会发生作用。图 5-9 说明了过滤器的概念。

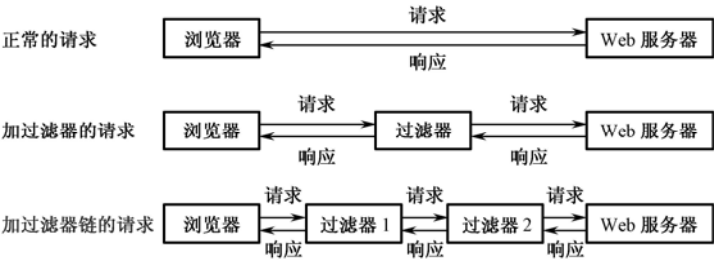


图 5-9 过滤器处理请求的过程

图 5-9 中显示了正常请求、加过滤器请求和加过滤器链请求的处理过程。过滤器可以对 `Request` 对象和 `Response` 对象进行处理。所有的过滤器类都必须实现 `java.Servlet.Filter` 接口，这个接口含有 3 个过滤器类必须实现的方法。

1) 过滤器必须实现的方法

(1) `init (FilterConfig)`。这是 `Servlet` 过滤器的初始化方法，`Servlet` 容器创建 `Servlet` 过滤器实例后将调用这个方法。在这个方法中可以通过 `FilterConfig` 参数读取 `web.xml` 文件中的 `Servlet` 过滤器的初始化参数。

(2) `doFilter (ServletRequest, ServletResponse, FilterChain)`。这个方法完成实际的过滤操作，当用户请求与过滤关联的 URL 时，Servlet 容器将先调用过滤器的 `doFilter` 方法，在返回响应之前也会调用此方法。`FilterChain` 参数用于访问过滤器链上的下一个过滤器。

(3) `destroy()`。Servlet 容器在销毁过滤器实例前调用该方法，这个方法可以释放 Servlet 过滤器占用的资源。

过滤器编写完成后，要在 `web.xml` 中进行配置，格式如下：

```
<filter>
    <filter-name>过滤器名称</filter-name>
    <filter-class>过滤器对应的类</filter-class>
    <!-- 初始化参数 -->
    <init-param>
        <param-name>参数名称</param-name>
        <param-value>参数值</param-value>
    </init-param>
</filter>
```

2) 过滤器的关联方式

过滤器必须和特定的 URL 关联才能发挥作用，过滤器的关联方式有 3 种：与一个 URL 关联、与一个 URL 目录下的所有资源关联、与一个 Servlet 关联。下面，在 `web.xml` 中配置过滤器与 URL 关联。

(1) 与一个 URL 资源关联：

```
<filter-mapping>
    <filter-name>过滤器名</filter-name>
    <url-pattern>xxx.jsp</url-pattern>
</ filter-mapping>
```

(2) 与一个 URL 目录下的所有资源关联：

```
<filter-mapping>
    <filter-name>过滤器名</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

(3) 与一个 Servlet 关联：

```
<filter-mapping>
    <filter-name>过滤器名</filter-name>
    <Servlet-name>Servlet 名称</Servlet-name>
</filter-mapping>
```

3) 过滤器完成的功能

前面讲述了过滤器的基本概念，那么过滤器有什么用处呢？常常利用过滤器完成以下功能：

- (1) 权限控制，通过过滤器实现访问的控制，当用户访问某个链接或者某个目录的时候，可以利用过滤器判断用户是否有访问权限。
- (2) 字符集处理，可以在过滤器中处理 `request` 和 `response` 的字符集，而不用在每个 Servlet 或者 JSP 中单独处理。
- (3) 其他一些场合，过滤器非常有用，可以利用它完成很多适合的工作，如计数器、数据加密、访问触发器、日志、用户使用分析等。

3. Struts2 配置

Struts2 配置可以分成几个单独的文件：web.xml、struts.properties、struts.xml。

其中 web.xml 是 Web 部署描述符，包括所有必需的框架组件；struts.xml 是 Struts2 的主要配置文件；struts.properties 是 Struts2 框架的属性配置文件。

1) web.xml

web.xml 并不是 Struts2 框架特有的文件。作为部署描述文件，web.xml 是所有 Java Web 应用程序都需要的核心配置文件。

Struts2 框架需要在 web.xml 文件中配置一个前端控制器 FilterDispatcher，用于对 Struts 框架进行初始化，以及处理所有的请求。

FilterDispatcher 是一个 Servlet 过滤器，它是整个 Web 应用的配置项。

2) struts.properties 文件

Struts2 提供了很多可配置的属性，通过这些属性的设置，可以改变框架的行为，从而满足不同 Web 应用的需求。这些属性可以在 struts.properties 文件中进行设置，struts.properties 是标准的 Java 属性文件格式，“#”号作为注释符号，文件内容由键（key）—值（value）对组成。

struts.properties 文件必须位于 classpath 下，通常放在 Web 应用程序的/WEB-INF/classes 目录下。

Struts2 在 default.properties 文件（位于 struts2-core-2.0.11.jar）中给出了所有属性的列表，并对其中一些属性设置了默认值。如果开发人员创建了 struts.properties 文件，那么在该文件中的属性设置会覆盖 default.properties 文件中的属性设置。

在开发环境中，以下几个属性是可能要修改的：

- (1) struts.i18n.reload=true，激活重新载入国际化文件的功能。
- (2) struts.devMode=true，激活开发模式，提供更全面的调试功能。
- (3) struts.configuration.xml.reload=true，激活重新载入 XML 配置文件的功能，当文件被修改后，就需要载入 Servlet 容器中的整个 Web 应用了。
- (4) struts2.url.http.port=8080，配置服务器运行的端口。

3) struts.xml 文件

struts.xml 是 Struts2 框架的核心配置文件，主要用于配置和管理开发人员编写的 Action。struts.xml 文件通常放在 Web 应用程序 WEB-INF/classes 目录下，在该目录下的 struts.xml 将被 Struts2 框架自动加载。

struts.xml 文件是一个 XML 文件，所以最开始的元素就是 XML 版本和编码信息。接下来则是 XML 的文档类型定义（DTD）。DTD 提供了 XML 文件中各个元素应使用结构的信息，而这些最终会被 XML 解析器或编辑器使用。

<struts>标签位于 Struts2 配置的最外层，其他标签都是包含在它里面的。

(1) package 元素。Struts2 中的包类似于 Java 中的包，提供了将 action、result、result 类型、拦截器和拦截器栈组织为一个逻辑单元的一种方式，从而简化了维护工作，提供了重用性。

与 Java 中的包不同的是，Struts2 中的包可以扩展另外的包，从而“继承”原来包的所有定义，并可以添加自己包特有的配置，以及修改原有包的部分配置，从这一点上看，Struts2 中的包更像 Java 中的类。

`package` 元素有一个必需的属性 `name`，指定包的名字，这个名字将作为引用该包的键。要注意的是，包的名字必须是唯一的，在一个 `struts.xml` 文件中不能出现两个同名的包。`package` 元素的 `extends` 属性是可选的，允许一个包继承一个或多个先前定义的包中的配置。`package` 元素的 `abstract` 属性是可选的，将其设置为 `true`，可以把一个包定义为抽象的。抽象包不能有 `Action` 定义，它只能作为“父”包被其他的包所继承。

【例 5-3】 `struts.xml` 文件中一个包定义的例子。

```
<?xml version= "1.0"encoding= "GBK"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundaion//DTD Struts Configuration 2.0//EN"
    http://struts.apache.org/dtds/struts-2.0.dtd>
<struts>
    <!-- 定义了一个名为 default 的包，继承自 struts-default.xml 文件中定义的
           struts-default 抽象包-->
    <package name= "default"extends= "struts-default">
        .....
    </package>
</struts>
```

(2) `namespace` 属性。`package` 元素的 `namespace` 属性可以将包中 `action` 配置为不同的命名空间。

当 Struts2 接收到一个请求的时候，它会将请求 URL 分为 `namespace` 和 `action` 名字两部分，Struts2 会从 `struts.xml` 中查找 `namespace/action` 这个命名对，如果没有找到，Struts2 就会在默认的命名空间中查找相应的 `action` 名。

默认的命名空间用空字符串 "" 来表示，若在定义包中没有使用 `namespace` 属性，那么就指定了默认的命名空间。

【例 5-4】 Struts2 文档中命名空间的例子。

```
<!-- -mypackage2 包在 /barspace 命名空间-->
<package name= "mypackage2"namespace= "/barspace">
    <action name= "bar"class= "mypackage.simpleAction">
        <result name= "success"type= "dispatcher">bar2.jsp</result>
    </action>
</package>
```

如果请求 `/barspace/bar.action`，框架将首先查找 `/barspace` 命名空间，如果找到了，则执行 `bar` `action`；如果没有找到，则到默认的命名空间中继续查找。在本例中，`/barspace` 命名空间中有名为 `bar` 的 `action`，因此它会被执行。

如果请求 `/barspace/foo.action`，框架会在 `/barspace` 命名空间中查找 `foo` 这个 `action`。如果找不到，框架会到默认命名空间中去查找。在本例中，`/barspace` 命名空间中没有 `foo` 这个 `action`，因此默认的命名空间中的 `/foo.action` 将被找到并执行。

如果请求 `/moo.action`，框架会在根命名空间 “/” 中查找 `moo` 这个 `action`，如果没有找到，再到默认命名空间中查找。

(3) `action` 元素。Struts2 的核心功能是 `Action`，对于开发人员来说，使用 Struts2 框架，主要的编码工作就是编写 `Action` 类。`Action` 类通常都是要实现 `com.opensymphony.`

xwork2.Action 接口，并实现该接口中的 execute()方法。

当然，Struts2 并不要求所编写的 Action 类一定要实现 Action 接口，也可以使用一个普通的 Java 类作为一个 action，只要该类提供一个返回类型为 String 的无参数的 public 方法即可。

在实际开发中，Action 类很少直接实现 Action 接口，通常都是从 com.opensymphony.xwork2.ActionSupport 类继承。ActionSupport 实现了 Action 接口和其他一些可选的接口，提供了输入验证、错误信息存取，以及国际化的支持，选择从 ActionSupport 继承，可以简化 Action 的开发。

开发好 action 后，就需要配置 action 映射，以告诉 Struts2 框架，针对某个 URL 的请求应该交由哪一个 action 进行处理。

当一个请求匹配某个 action 的名字时，框架就使用这个映射来确定如何处理请求。

```
<action name="login"class= "org.apex.struts2.action.LoginAction">
    <result>/success.jsp</result>
    <result name= "error">/error.jsp</result>
</action>
```

class 属性是 Action 实现类的完整类名。在执行 action 时，默认调用的是 execute()方法。例如，上面例子中请求 login.action 时，将调用 org.apex.struts2.action.LoginAction 实例的 execute()方法。

也可以通过 action 元素的 method 属性来指定 action 所指定的方法。在 struts.xml 文件中，可以为同一个 Action 类配置不同的别名，并使用 method 属性。

【例 5-5】通过 action 元素的 method 属性来指定 action 所指定的方法。

```
<package name= "default"extends= "struts-default">
    <!-- 请求/list,调用 NewsAction 的 execute 方法-->
    <action name= "list"class= "org.apex.struts2.action.NewsAction">
        <result>/listNews.jsp</result>
    </action>
    <!-- 请求/create,调用 NewsAction 的 create 方法-->
    <action name="create"class= "org.apex.struts2.action.NewsAction"method=
        "create">
        <result type= "redirect">/list.action</result>
    </action>
    <!-- 请求/edit,调用 NewsAction 的 edit 方法-->
    <action name= "delete"class= "org.apex.struts2.action.NewsAction"method=
        "delete">
        <result type= "redirect">/list.action</result>
    </action>
</package>
```

(4) result 元素。一个 result 代表了一个可能的输出。当 Action 类的方法执行完成时，它返回一个字符串类型的结果代码，框架根据这个结果代码选择对应的 result，向用户输出。

在 com.opensymphony.xwork2.Action 接口中定义了一组标准的结果代码，可提供开发人员使用，代码如下所示：

```
public interface Action{
    public static final String SUCCESS = "success";
```

```

public static final String NONE = "none";
public static final String ERROR = "error"
public static final String INPUT = "input";
public static final String LOGIN = "login";
}

```

除了这些预定义的结果代码以外，开发人员也可以定义其他的结果代码来满足自身应用程序的需要。

【例 5-6】 开发人员定义其他的结果代码。

```

<package name= "default"extends= "struts-default">
    <action name= "login">
        <result>/success.jsp</result>
        <result name = "error">/error.jsp</result>
    </action>
</package>

```

5.2.4 Struts2 的Action

1. 概述

Struts2 控制器最重要的组成部分是 Action，它是 Web 框架的控制中心，是连接模型和视图的桥梁和纽带，如图 5-10 所示。

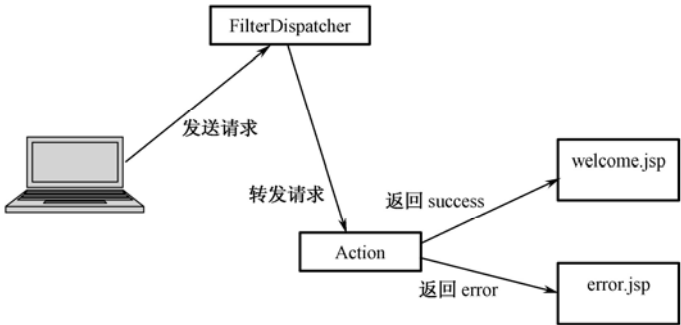


图 5-10 应用的处理流程

前面所讲到的 FilterDispatcher 是前端控制器、一个 Servlet 过滤器，用于对 Struts 框架进行初始化，以及转发所有的请求。Action 接收来自视图层的请求，并接收请求参数，同时负责调用模型层方法来完成业务逻辑的处理，最后控制程序的流程，选择一个合适的视图将结果显示给客户（例如，例 5-1 所开发的一个用户登录 Web Project，如果输入姓名和密码 success，则进入欢迎页面；反之，进入失败页面）。

2. Action的定义

Struts2 中的 Action 只需要在一个普通的类中定义一个方法，例如：

```

public class XXXAction{
    public String method() {
        return "return Value";
    }
}

```

XXXAction: Action 的类名，习惯上以 Action 结尾，更容易阅读和理解。

method: 用于接收请求的方法，名称可以自定义，默认情况下，会调用 `execute()` 方法。该方法不能带任何参数，且必须返回字符串类型。

return: 返回值，类型必须是字符串，Struts2 会根据返回值控制程序流程。

【例 5-7】 定义一个 `HelloAction`，访问该 Action 的时候在控制台打印“你好”。

```
public class HelloAction{
    public String hello(){
        System.out.println("你好");
        return null;
    }
}
```

在 `struts.xml` 配置文件中，将刚才创建的 Action 注册到这里。

```
<action name= "helloAction"class= "HelloAction"method= "hello">
</action>
```

name: 自定义名称，访问 Action 时用到，如 `helloAction.action`。

class: Action 的类名。

method: 访问该 Action 时，调用 `hello()` 方法。

因为 `hello()` 方法的返回值是 `null`，表示不跳转到任何地方。如果返回一个字符串，则必须配置 `<action>` 的子标签 `<result>`，通过该标签映射一个跳转路径，如 `<result name="success">/my.jsp</result>`，表示如果方法的返回值为 `success`，则跳转到 `my.jsp` 文件。

`<method>` 是可选的，在定义 Action 的响应方法时，如果将方法名称定义为 `execute`，则 `<method>` 可省略。

【例 5-8】 方法名称定义为 `execute` 的 `HelloAction` 类。

```
public class HelloAction{
    public String execute(){
        System.out.println( "你好");
        Return null;
    }
}
```

配置修改如下：

```
<action name= "helloAction" class= "HelloAction">
</action>
```

3. 通过 Action 获取请求参数

通过 `HttpServletRequest` 的 `getParameter()` 或 `getParameterValues()` 方法可以得到从客户端传送过来的请求参数，但是这很麻烦，而且增加了应用程序的耦合度，增强了对容器的依赖。Struts2 在 Action 中改进了获取请求参数的方式，自动获取请求参数。

Struts2 获取请求参数的名称，拼成该参数的 `set` 方法和 `get` 方法，调用方法实现属性的存取操作。例如，从客户端传送一个名叫 `name` 的参数，则会拼成 `setName` 和 `getName` 方法名，通过反射调用 `setName()` 方法进行赋值，程序员通过 `getName()` 方法就能取到值了。存取值的代码写在 Action 中就可以。上面的操作是交给 OGNL 实现的。

以下是一个 Action 获取请求参数的例子。本示例用于演示用户登录的过程。用户输入用户名和密码，如果分别为 `admin` 和 `admin`，则显示登录成功的信息，否则显示登录失败的

信息。

【例 5-9】 一个 Action 获取请求参数的例子。

...

```
<form action= "loginAction.action"method= "post">
    用户名: <input name= "username"><br>
    密码: <input name= "password"type= "password"><br>
    <input type= "submit"value= "登录">
</form>
```

在表单中，表单域的 **name** 属性值必须和 **Action** 中定义的属性名称一致，才能正确地被 **Action** 接收，如果读者懂得反射原理，更容易理解。

LoginAction.java 作为控制器，负责接收视图层发送过来的用户名和密码，并通过 **execute()** 方法调用业务方法，根据执行结果控制程序流程：如果登录成功，则跳转到 **success.jsp**，否则跳转到 **failure.jsp**。

```
public class LoginAction{
    private String username;
    private String password;
    public void setPassword(String password) {
        this.password = password;
    }
    public String getPassword() {
        return password;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getUsername() {
        return username;
    }
    public String execute(){
        //调用业务组件，如果成功，返回“success”；否则，返回“failure”。
        if(...){
            return "success";
        }else{
            return "failure";
        }
    }
}
```

提交表单后，请求提交给 **loginAction.action**。**loginAction.action** 是在 **struts.xml** 文件中预先配置好的，可以自定义，但最好用一个比较有意义的名字，增强程序的易读性。

LoginAction 的配置：

```
<action name= "loginAction"class= "LoginAction">
    <return name= "success">/success.jsp</result>
    <return name= "failure">/ failure.jsp</result>
```

</action>

也可以将定义在 Action 中的属性封装成一个实体类，更利于控制器与属性的分离。将属性定义在 Action 中的做法，违背了控制器的初衷。Struts2 的开发者们提供了一种更好的处理方式，类似于 struts1.x 中的 ActionForm，但是比 ActionForm 更加灵活。

【例 5-10】将定义在 Action 中的属性封装成一个实体类。

User.java 代码：

```
public class User{
    private String username;
    private String password;
    public void setPassword(String password) {
        this.password = password;
    }
    public String getPassword() {
        return password;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getUsername() {
        return username;
    }
}
```

在 LoginAction 类中，保存一个 User 的引用即可，该类的配置不需要改变。

LoginAction.java 的修改版的代码如下：

```
public class LoginAction{
    private User user;
    public void setUser(User user){
        this.user = user;
    }
    public User getUser(){
        return user;
    }
    public String execute(){
        //同上
    }
}
```

最后，修改 login.jsp 中表单域的 name 属性值，基本格式为：引用名.属性名。在上例中，引用名是指定义在 LoginAction 类中的 user，属性名是指定义在 Users 中的 username 和 password。为了将用户名传递到 User 类的 username 属性中保存，修改<input name = "username">为<input name = "user.username">即可，所以 login.jsp 最后的代码如下：

<body>

<form action= "loginAction.action"method = "post">


```

用户名: <input name= "user.username"><br>
密码: <input name= "user.password" type= "password"><br>
<input type= "submit" value= "登录">
</form>
</body>

```

4. ActionSupport

在 Struts2 中, Action 与容器已经做到完全解耦, 不再继承某个类或实现某个接口, 但是在特殊情况下, 为了降低编程的工作难度, 充分利用 Struts2 提供的功能, 定义 Action 时会继承类 ActionSupport, 该类位于 XWork2 提供的包 com.opensymphony.xwork2 中。

ActionSupport 类为 Action 提供了一些默认实现, 主要包括:

- (1) 预定义常量;
- (2) 从资源文件中读取文本资源;
- (3) 接收验证错误信息;
- (4) 验证的默认实现。

【例 5-11】 ActionSupport 类所实现的接口。

```

public class ActionSupport implements Action, Validateable, ValidationAware,
    TextProvider, LocaleProvider, Serializable{
}

```

Action 接口同样位于 com.opensymphony.xwork2 包, 定义了一些常量和 execute()方法。

```

public interface Action{
    public static final String SUCCESS = "success";
    public static final String NONE = "none";
    public static final String ERROR = "error";
    public static final String INPUT = "input";
    public static final String LOGIN = "login";
    public String execute() throws Exception;
}

```

Action 接口中一共定义了 5 个常量, 每个常量都有特定的意义, 这些常量被 execute()方法返回, 并最终被 Result 处理, <action>的子标签<result>的 name 属性可以是这些常量中的任何一个。其中, success 是 name 属性的默认值, 表示请求处理成功; error 表示请求处理失败, none 表示请求处理完成后不跳转到任何页面, input 表示输入时如果验证失败应该跳转到什么地方; login 表示登录失败后跳转的目标。

接口 com.opensymphony.xwork2.ValidationAware 的实现类 com.opensymphony.xwork2.ValidationAwareSupport 定义了 3 个集合成员, 这些集合用于存储错误或消息。ValidationAware 的众多方法主要完成对这些成员的存储操作, 并判断集合中是否有元素, ActionSupport 仅仅实现对这些方法的简单调用。

5.2.5 Struts2 的OGNL表达式

1. Struts2 的OGNL表达式

OGNL 是 Object Graphic Navigation Language (对象图导航语言) 的缩写, OGNL 是一

个开源项目。OGNL 是一种功能强大的 EL (Expression Language, 表达式语言), 可以通过简单的表达式来访问 Java 对象中的属性。

OGNL 先在 WebWork 项目中得到应用, 也是 Struts2 框架视图默认的表达式语言, 可以说, OGNL 表达式是 Struts2 框架的特点之一。

标准的 OGNL 会设定一个根对象 (root 对象)。假设使用标准 OGNL 表达式来求值 (不是 Struts2 OGNL), 如果 OGNL 上下文有两个对象: foo 对象和 bar 对象, 同时 foo 对象被设置为根对象(root), 则利用下面的 OGNL 表达式求值。

【例 5-12】用 OGNL 表达式求值。

```
#foo.blah           //返回 foo.getBlah()  
#bar.blah           //返回 bar.getBlah()  
blah                //返回 foo.getBlah(), 因为 foo 为根对象
```

使用 ONGL 非常简单, 如果要访问的不是根对象, 如示例中的 bar 对象, 则需要使用命名空间, 用 “#” 来表示, 如 “#bar”; 如果访问一个根对象, 则不用指定命名空间, 可以直接访问根对象的属性。

在 Struts2 框架中, 值栈 (Value Stack) 就是 OGNL 的根对象, 假设值栈中存在两个对象实例: Man 和 Animal, 这两个对象实例都有一个 name 属性, Animal 有一个 species 属性, Man 有一个 salary 属性, 假设 Animal 在值栈的顶部, Man 在 Animal 后面, 如图 5-11 所示。

【例 5-13】理解 OGNL 表达式。

```
species //调用 animal.getSpecies()  
salary  //调用 man.getSalary()  
name    //调用 animal.getName(), 因为 Animal 位于值栈的顶部
```

最后一行实例代码中, 返回的是 animal。getName()返回值返回 Animal 的 name 属性, 因为 Animal 是值栈的顶部元素, OGNL 将从顶部元素搜索, 所以会返回 Animal 的 name 属性值。如果要获得 Man 的 name 值, 则需要如下代码:

```
man.name
```

Struts2 允许在值栈中使用索引, 实例代码如下所示:

```
[0].name //调用 animal.getName()  
[1].name //调用 man.getName()
```

Struts2 中的 OGNL Context 是 ActionContext, 其结构如图 5-12 所示。

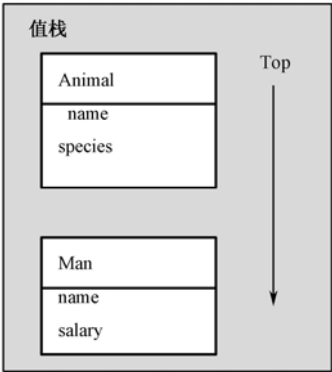


图 5-11 一个包含了 Animal 和 Man 的值栈

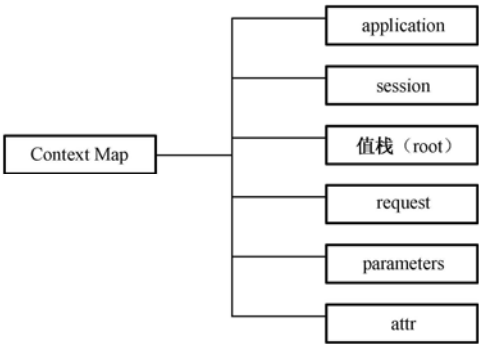


图 5-12 Struts2 的 OGNL Context 结构示意图

由于值栈是 Struts2 中的 OGNL 的根对象，如果用户需要访问值栈中的对象，则可以通过如下代码访问值栈中的属性：

```
${foo} //获得值栈中的foo属性
```

如果访问其他 Context 中的对象，由于不是根对象，在访问时需要加#前缀。

【例 5-14】访问不是根对象的需要加#前缀的其他 Context 中的对象。

(1) application 对象，用于访问 ServletContext，例如，#application.userName 或者 #application["username"]，相当于调用 Servlet 的 getAttribute("username")。

(2) session 对象，用于访问 HttpSession，例如，#session.userName 或者 #session["username"]，相当于调用 session.getAttribute("username")。

(3) request 对象，用于访问 HttpServletRequest 属性的 Map，例如，#request.userName 或者 #request["username"]，相当于调用 request.getAttribute("username")。

2. OGNL的集合操作

如果需要一个集合元素（如 List 对象或者 Map 对象），可以使用 OGNL 中同集合相关的表达式。使用如下代码直接生成一个 List 对象：

```
{e1,e2,e3,...}
```

该 OGNL 表达式中，直接生成了一个 List 对象，该 List 对象中包含 3 个元素：e1、e2 和 e3。如果需要更多的元素，可以定义多个元素，多个元素之间使用逗号隔开。

如下代码可以直接生成一个 Map 对象：

```
#{key:value1,key2:value2,...}
```

Map 类型的集合对象使用 key-value 格式定义，每个 key-value 元素使用冒号表示，多个元素之间使用逗号隔开。

对于集合类型，OGNL 表达式可以使用 in 和 not in 符号。其中，in 判断某个元素是否在指定的集合对象中；not in 判断某个元素是否不在指定的集合对象中。

【例 5-15】集合类型 OGNL 表达式。

```
<s:if test= " 'foo' in{'foo', 'bar'}">
...
</s:if>
```

除了 in 和 not in 之外，OGNL 还允许使用某个规则获得集合对象的子集，常用的有以下 3 个相关操作符。

(1) ? 获得所有符合逻辑的元素。

(2) ^ 获得符合逻辑的第一个元素。

(3) \$ 获得符合逻辑的最后一个元素。

例如：

```
Person.relatives.{?# this.gender= 'male'}
```

该代码可以获得 person 的所有性别为 male 的 relatives 集合。

5.2.6 Struts2 的标签库

Struts2 提供了许多不同种类的标签，可以分为 4 类：数据标签、控制流标签、UI 标签及杂项标签。

数据标签处理从值栈提取数据，以及将数据设置到值栈中的操作。

控制流标签可以改变程序的执行流，以及基于系统的状态产生不同的输出。

杂项标签难以准确分类，但也很有用。

1. 数据标签

1) <s:property>标签

property 标签得到“value”属性，在 Action 中为 user、username 属性赋值，在网页中从 user 中读取值。代码如下所示：

```
<s:property value = "user.username"/>
```

2) <s:set>标签

set 标签用于对值栈中的表达式进行求值，并将结果赋给特定作用域中的某个变量名。这对于在 JSP 中使用临时变量是很有作用的。而使用临时变量会使代码更容易阅读，并使执行稍微快一点。

下面是一个简单例子，展示了 property 标签访问存储于 session 的 user 对象的多个字段：

```
<s:property value = "#session['user'].username"/>
```

```
<s:property value = "#session['user'].age"/>
```

```
<s:property value = "#session['user'].address"/>
```

每次都要重复使用#session['user']不仅麻烦，还容易引发错误。更好的做法是定义一个临时变量，让这个变量指向 User 对象，使用 set 标签使得代码易于阅读。

【例 5-16】<s:set>标签。

```
<s:set name= "user" value= "#session['user']" />
```

```
<s:property value= "#user.username"/>
```

```
<s:property value= "#user.age"/>
```

```
<s:property value= "#user.address"/>
```

由于 set 标签可以将表达式重构得更精简，更易于管理。因而，整个页面都变得更简单了。

3) <s:bean>标签

基本的 Struts2 标签提供了一定的数据处理功能，而有时候需要更加复杂的功能，bean 标签可以创建简单的 JavaBean，并将其压入值栈中，在 bean 标签的起始与结束标记之间，还可以任意地把 JavaBean 赋值给某个变量名，以便让它在 action context 中能够访问。

【例 5-17】<s:bean>标签。

Counter bean 用于跟踪计数。

```
<s:bean name= "com.opensymphony.webwork.util.Counter" id= "counter">
```

```
    <s:param name= "last" value= "100">
```

```
</s:bean>
```

```
<s:iterator value= "#counter">
```

```
    <s:property/>
```

```
</s:iterator>
```

在这个例子中，首先 Counter bean 被创建，接着以 100 为参数调用 setLast()方法，然后使用 iterator 标签对其循环取值，而每次循环得到的值将被打印出来。

4) <s:action>标签

有时候，bean 标签还不足以实现复杂的或者可重用的视图。在 JSP 中执行 action 并访问相应的数据，而不是将 JavaBean 存入 action context 中。

利用 action 标签，可以通过简单的方式创建可重用组件，同时不需要在 JSP 页面增加代

码片段。例如，在应用系统中，页面左边是书的种类的菜单，由于需要在多个页面进行显示，所以创建一个独立获取数据的 `action` 供各个页面使用。

```
<action name= "browseCatalog" executeResult= "true">
    <result name= "success">/menu.jsp</result>
</action>
```

`executeResult` 设置为 `true`。如果 `executeResult` 没有设定，在默认情况下，它的值为 `false`，即使 `action` 执行了，也不会生成任何视图。

2. 控制流标签

(1) `<s:if><s:else>` 标签：执行基本的条件流转。例如，判断用户是否登录，如果登录，页面显示“注销”；反之，页面显示“登录”。

【例 5-18】 判断用户是否登录的 `<s:if><s:else>` 标签。

```
<s:if test= "#session.user= =null">
    <a class=title01 href= "login.jsp">登录</a>
</s:if>
<s:else>
    <a href= "logout.action">注销</a>
</s:else>
```

(2) `<s:iterator>` 标签。`iterator` 标签可以循环遍历任何对象集合，包括 `Collection`、`Map`、`Enumeration`、`Iterator` 及 `Array`。同时，可以在 `action context` 中定义一个变量，用于确定与当前循环状态相关的基本信息，例如，遍历到了奇数行还是偶数行。

下面是一个例子，用于循环遍历由 `CaveatEmptor` 的 `Search action` 返回的条目集合。

【例 5-19】 用于循环遍历的 `<s:iterator>` 标签。

```
<s:iterator value= "items">
    <s:property value= "name"/>,<s:property value= "description"/>
</s:iterator>
```

表达式 `items` 调用了 `Search.getItems()` 方法，执行后返回一个 `Item` 对象的 `List`。随着循环遍历的进行，在 `iterator` 标签内部的内容被调用的时候，每个遍历到的对象都会被暂时压入值栈。在标签内部的内容执行完毕后，这个对象就会出栈。

由于 `Item` 对象被压入到栈中，所以 `property` 标签能够通过使用 `name` 和 `description` 这两个表达式，实现 `getName()` 和 `getDescription()` 方法。

5.3 Hibernate 框架

`Hibernate` 是一个开放源代码的对象关系映射框架，它对 `JDBC` 进行了轻量级的对象封装，使 `Java` 程序员可以随心所欲地使用对象编程思维来操纵数据库。它不仅提供了从 `Java` 类到数据表之间的映射，也提供了数据查询和事务机制。相对于使用 `JDBC` 和 `SQL` 来手工操作数据库，`Hibernate` 可以大大减少操作数据库的工作量。另外，`Hibernate` 可以利用代理模式来简化载入类的过程，这将大大减少利用 `Hibernate QL` 从数据库提取数据的代码编写量，从而节约开发时间和开发成本。`Hibernate` 可以和多种 `Web` 服务器或者应用服务器良好集成，如今已经支持几乎所有流行的数据库服务器。

本节将较详细论述 Hibernate 的理论基础、案例、实用工具和 Hibernate 的配置、高级特性等。

5.3.1 Hibernate概述

对象关系映射（ORM，Obiect Relational Mapping）是一种为了解决面向对象与关系数据库存在的互不匹配现象的技术。简单地说，ORM 是通过使用描述对象和数据库之间映射的元数据，将 Java 程序中的对象自动持久化到关系数据库中。本质上就是将数据从一种形式转换到另外一种形式。对象和关系数据是业务实体的两种表现形式，业务实体在内存中表现为对象，在数据库中表现为关系数据。内存中的对象之间存在关联和继承关系，而在数据库中，关系数据无法直接表达多对多关联和继承关系。因此，对象—关系映射系统一般以中间件的形式存在，主要实现程序对象到关系数据库数据的映射。

面向对象是从软件工程基本原则（如耦合、聚合、封装）的基础上发展起来的，而关系数据库则是从数学理论发展而来的，两套理论存在显著的区别。为了解决这个不匹配的现象，对象关系映射技术应运而生。

目前流行的各种 ORM 框架较多，如 Java 阵营的 Hibernate、iBatis、JDO、微软的 ObjectSpaces，DevExpress 公司的 XPO 等。Hibernate 是目前最流行的 ORM 开发工具。

1. Hibernate的体系结构

Hibernate 作为 ORM 开发工具，通过配置文件 hiberante.cfg.xml 或 hibernate.properties 和映射文件 (*.hbm.xml) 把 Java 对象或持久化对象（PO，Persistent Obeject）映射到数据库中的数据表，然后通过操作 PO，对数据库中的表进行各种操作。

Hibernate 的体系结构，如图 5-13 所示。

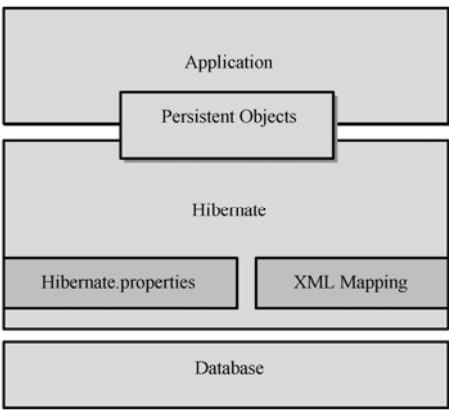


图 5-13 Hibernate 的体系结构

Hibernate 使用了 Java EE 架构中的一些技术，如 JDBC、JTA（Java Transaction API，Java 关于数据库事务的 API）和 JNDI。其中，JDBC 是一个支持关系数据库操作的基础层，它与 JNDI 和 JTA 结合在一起，使得 Hibernate 可以方便地集成到 Java EE 应用服务器中去。

下面介绍 Hibernate 的主要接口及功能。

（1）Session 接口：一个持久层管理器，是一个轻量级的类。它包含一些持久层相关的

操作，如存储持久对象至数据库，以及从数据库中获得它们，不同于 JSP 应用中的 HttpSession。

(2) SessionFactory 接口：工厂模式，用户程序从工厂类 SessionFactory 中取得 Session 的实例。

(3) Configuration 接口：该接口定位映射文档的位置，读取这些配置，然后创建一个 SessionFactory 对象。

(4) Transaction 接口：对实际事务实现的一个抽象，如对 JDBC、JTA、CORBA 事务。

(5) Query 和 Criteria 接口：对数据库及持久对象进行查询的接口。

(6) Callback 接口：当一些有用的事件发生时，该接口会通知 Hibernate 去接收一个通知消息，可用于日志。

2. Hibernate的文档和软件

在 Hibernate 的官方网站 (<http://www.hibernate.org>) 可以下载最新的 Hibernate 包，其简单介绍如表 5-3 所示。下载最新的版本之后，解压缩 Hibernate 包中有一个 hibernate.jar 和 lib 目录。在 lib 目录中包括了许多 JAR 文件，如 dom4j、CGLIB、asm、Commons Collections、Commons Logging 和 EHCACHE 等。这些 JAR 文件有一些在运行 Hibernate 时是必需的。

表 5-3 Hibernate 包的简单介绍

包 名	包 的 作 用
hibernate3.jar	核心框架包
cglib-asm.jar	CGLIB 库，Hibernate 用它来实现 PO 字节码的动态生成，非常核心的库，必须使用的 jar 包
dom4j.jar	dom4j 是一个 Java 的 XML API，类似于 jdom，用来读写 XML 配置文件
commons-collections.jar	包含了一些 Apache 开发的集合类，功能比 java.util.*强大，必须使用的 jar 包
commons-ang.jar	包含了一些数据类型工具类，是 java.lang.*的扩展，必须使用的 jar 包
commons-logging.jar	包含了日志功能，必须使用的 jar 包。这个包本身包含了一个 Simple Logger，但是功能很弱。在运行的时候它会先在 CLASSPATH 中寻找 log4j
antlr-2.7.6.jar	Hibernate 使用 ANTLR 来产生查询分析器，这个类库在运行环境下时也是必需的
jta.jar	当 Hibernate 使用 JTA 的时候才需要
ehcache-1.1.jar	Hibernate 可以使用不同 cache 缓存工具作为二级缓存。EHCACHE 是默认的 cache 缓存工具
dom4j-1.6.1.jar	Hibernate 使用 dom4j 解析 XML 配置文件和 XML 映射的元文件

5.3.2 Hibernate的运行及其映射、基本配置和接口

1. Hibernate的环境、映射、基本配置与运行

1) Hibernate 的环境

Hibernate 可以运行于单机之上，也可以运行于 Web 应用程序之中。如果运行于单机，则将所有用到的 jar 包（包括 JDBC 驱动程序）包含到 classpath 中；如果运行于 Web 应用程序中，则将 jar 包放置于 WEB-INF/lib 中。本节中的实例将 Hibernate 作为应用程序运行于单机中。

Hibernate 运行时，先从 Hibernate 配置文件中读取数据库的连接信息，进行数据库连接，然后通过映射文件动态建立持久化对象实例。这些实例就是数据记录。由于 Hibernate API 对 JDBC 做了封装，所以，利用 Hibernate API 可以很方便地操作持久化类，从而达到

和直接使用 JDBC 操作数据库一样的效果。

要运行本节中的案例，必须安装一个数据库服务器，并且有相关的 JDBC 驱动程序。本节将使用 MySQL 5.0 数据库，假设读者已经掌握了 JDBC 的基本知识，并能使用 MySQL 的 JDBC 驱动程序。本节将使用从 MySQL 官方网站上下载驱动程序 mysql-connector-java-3.1.12-bin.jar。

要运行于单机上，必须涉及 classpath 的设置。将图 5-14 中的包置于 classpath 中。假设将把所涉及的 jar 文件放置在 d:\jars 目录中，如图 5-14 所示。

如果读者使用集成 IDE，可以将这些包导入外加类库中即可。如果读者使用 JDK 5.0，用命令行方式运行程序，可在系统环境变量或系统自动批处理文件中来设置 classpath。另外，也可以在 DOS 窗口下直接进行设置（本 DOS 窗口有效）。

接下来可以将 hibernate3.zip 解压后在 etc 目录下的 log4j.properties 复制到 Hibernate 项目的 classpath 下，并修改其中的 log4j.logger.org.hibernate 为 error，即只在错误发生时显示必要的信息。

下面讲解案例中文件的具体布置和配置，以及运行情况。先建立工作目录 \chap05\hbexample1，再建立子目录 liufy，如 D:\chap05\hbexample1\liufy。子目录 liufy 中的文件如图 5-15 所示。

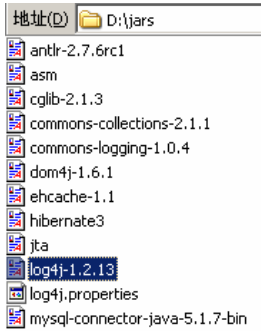


图 5-14 classpath 中涉及的包

名称	大小	类型
FirstHibernate.class	1 KB	CLASS 文件
User.class	1 KB	CLASS 文件
User.hbm	1 KB	XML 文档

图 5-15 简单案例文件布置

接下来，设置环境变量，编写批处理文件，如 setjars.bat，如图 5-16 所示。然后打开 Dos 窗口，运行 setjars.bat。假设 D:\jars 目录下存放了所有可能用到的包。

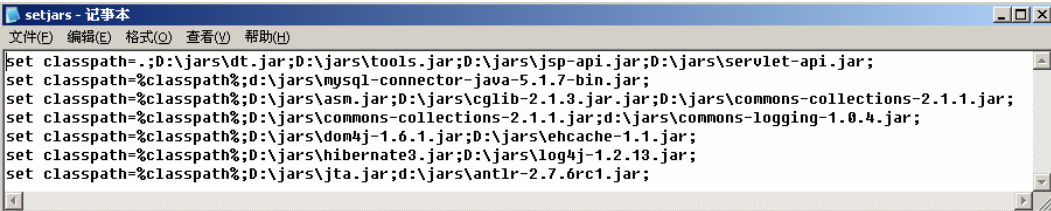


图 5-16 系统环境变量的设置

如果使用 JDK 6.0，那么还可以使用 JDK 6.0 的新功能来设置 classpath:

```
set classpath = . ; d:\ jars\ *
```

2) Hibernate 的映射

下面以一个简单的单机程序来示范 Hibernate 的映射与运行。首先做数据库的准备工作，在 MySQL 中新增一个 hdata1 数据库，并建立 user 表，其内容如下（CreateUser.sql）:

```
CREATE TABLE user (id INT(11) NOT NULL auto_increment PRIMARY KEY,
                    name VARCHAR(100) NOT NULL default "",
                    age INT);
```


对于这个表，必须有一个 User 类与之对应，表中的每一个字段将对应至 User 实例上的 Field 成员。这个 User.java 类就是一个 POJO。

POJO 在 Hibernate 语义中理解为数据库表所对应的 Domain Object。这里的 POJO 就是所谓的“Plain Ordinary Java Object”。从字面上来讲，就是无格式普通 Java 对象。简单地可以理解为一个不包含逻辑代码的值对象（Value Object，VO）。

【例 5-20】与 user 表对应的 User.java 类。

```
package liufy;

public class User {
    private Integer id;
    private String name;
    //必须有一个预先设置的构造方法
    public User() { }
    public Integer getId() {return id; }
    public void setId(Integer id) {this.id = id; }
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
    public Integer getAge() {return age;}
    public void setAge(Integer age) {this.age = age;}
}
```

其中，id 是一个特殊的属性，Hibernate 会使用它来作为主键识别，这是在 XML 映像文件中完成的。为了告诉 Hibernate 用户所定义的 User 实例如何映像至数据库表，下面编写一个 XML 映射文件。

【例 5-21】编写一个 XML 映射文件，文档名为 User.hbm.xml。

```
<? xml version = "1.0" encoding = "utf-8"? >
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name = "liufy.User" table = "user">
<id name = "id" column = "id" type = "java.lang.Integer">
<generator class = "native"/>
</id>
<property name = "name" column = "name" type = "java.lang.String"/>
<property name = "age" column = "age" type = "java.lang.Integer"/>
</class>
</hibernate-mapping>
```

一般的 Hibernate.hbm.xml 映射文件主要包括 3 部分内容：表名一类名映射、主键映射、字段映射。

(1) 表名一类名映射：<class>标签中的 name 属性为映像的对象，而 table 为映像的表。

(2) 主键映射：<id>中 column 属性指定了表的 id 字段，而 type 属性指定了 User 实例中 id 字段的数据类型，这时 type 中设定的是直接指定 Java 中的对象类型，Hibernate 也定义有自己的映像类型，作为 Java 对象与 SQL 数据的标准对应类型。type = “java.lang.Integer”表示当前字段的数据类型为 Integer。id 是一个特殊的属性，代表了这个类的数据库标识符

(主键)，它对于类似于 User 这样的实体是必需的。这样，在 User 类和 user 表之间建立了基于 id 进行识别的唯一性能映射关系。Hibernate 将根据 User 对象的 id 属性确定与之对应的库表记录。

`<generator class="native"/>`指定了主键生成方式。`<id>`中主键的产生方式在这里设定为 `generator class = native`，表示主键的生成方式是由 Hibernate 根据数据库 Dialect 的定义来决定的，当然还有其他主键的生成方式。对于不同的数据库和应用程序，主键生成方式往往不同。有的情况下，主键由应用逻辑生成。

Hibernate 的主键策略分为 3 大类：Hibernate 对主键 id 赋值；应用程序自己对 id 赋值；由数据库对 id 赋值。下面分别介绍 Hibernate 的属性值。

assign: 应用程序自己对 id 赋值。当设置 `<generator class="assigned"/>` 时，应用程序需要自己负责主键 id 的赋值。例如，下述代码：

```
User user = new User();
user.setId(new Integer(200001212));
user.setUsername("tom");
user.setPassword("tom");
session.save(user);
```

native: 由数据库对 id 赋值。当设置 `<generator class="native"/>` 时，数据库负责主键 id 的赋值，最常见的是 int 型的自增型主键。假如，在 MySQL 中建立表的 id 为 auto_increment，则应用程序的编码如下：

```
User user = new User();
user.setUsername("Tom");
user.setPassword("Tom");
session.save(user);
```

hilo: 通过 hi/lo 算法实现的主键生成机制，需要额外的数据库表保存主键生成历史状态。

seqhilo: 与 hilo 类似，通过 hi/lo 算法实现的主键生成机制，只是主键历史状态保存在 Sequence 中，适用于支持 Sequence 的数据库，如 Oracle。

increment: 主键按数值顺序递增。此方式的实现机制为在当前应用实例中维持一个变量，以保存当前的最大值，之后每次需要生成主键的时候将此值加 1 作为主键。这种方式可能产生的问题是：如果当前有多个实例访问同一个数据库，由于各个实例各自维护主键状态，不同实例可能生成同样的主键，从而造成主键重复异常。因此，如果同一个数据库有多个实例访问，这种方式应该避免使用。

identity: 采用数据库提供的主键生成机制，如 SQL Server、MySQL 中的自增主键生成机制。

sequence: 采用数据库提供的 sequence 机制生成主键，如 Oracle Sequence。

native: 由 Hibernate 根据数据库适配器的定义，自动采用 identity、hilo、sequence 的其中一种作为主键生成方式。

uuid.hex: 由 Hibernate 基于 128 位唯一值产生算法，根据当前设备 IP、时间、JVM 启动时间、内部自增量等 4 个参数生成十六进制数值（编码后长度为 32 位的字符串表示）作为主键。即使是在多实例并发运行的情况下，这种算法也能在最大限度上保证产生的 id 的唯一性。当然，重复的概率在理论上依然存在，只是概率比较小。一般而言，利用 uuid.hex 方式生成主键将提供最好的数据插入性能和数据平台适应性。

uuid.string: 与 **uuid.hex** 类似，只是生成的主键进行编码（长度 16 位），在某些数据库中可能出现問題。

foreign: 使用外部表的字段作为主键。

select: Hibernate 3 中新引入的主键获取机制，主要针对的是遗留系统的改造工程。

由于常用的数据库，如 SQL Server、MySQL 等，都提供了易用的主键生成机制（如 **auto-increase** 字段），可以在数据库提供的主键生成机制上，采用 **generator-class=native** 的主键生成方式。

（3）属性、字段映射：属性、字段映射将映射类属性与库表字段相关联。同样地，例 5-21 的 `<property>` 标签中的 **column** 与 **type** 都各自指明了表中字段与对象中属性的对应。Hibernate 对属性使用的类型不加限制。所有的 Java JDK 类型和原始类型（如 **string**、**char** 和 **float**）都可以被映射，也包括 Java 集合框架（Java Collections Framework）中的类，还可以把它们映射成为值、值集合，或者与其他实体相关联。例如：

```
<property name= "username" type= "java.lang.String">    //属性、字段映射
    <column name = "username" length= "1.0" not-null= "true"/>
</property>
name="username" //指定了映射类中的属性名为“username”，此属性将被映射到指定的库表字段
type= "java.lang.String" //指定了映射字段的数据类型
column name ="username" //指定库表中对应映射类属性的字段名，如字段名和属性名均为
“username”
```

这样，就将 User 类的 **username** 属性和库表 **user** 的 **username** 字段相关联。Hibernate 将从 **user** 表中 **username** 字段读取的数据作为 User 类的 **username** 属性值。同样在进行数据保存操作时，Hibernate 将 User 类的 **username** 属性写入 **user** 表的 **username** 字段中。

Hibernate 从本质上来讲是一种“对象—关系型数据映射”（ORM，Object Relational Mapping）。前面的 POJO 在这里体现的就是 ORM 中 Object 层的语义，而映射（Mapping）文件则是将对象（Object）与关系型数据（Relational）相关联的纽带。在 Hibernate 中，映射文件通常以“**.hbm.xml**”作为后缀。

3）Hibernate 的基本配置

Hibernate 同时支持 XML 格式的配置文件，以及传统的 **properties** 文件配置方式。不过，这里建议采用 XML 型配置文件。XML 配置文件提供了更易读的结构和更强的配置能力，可以直接对映射文件加以配置，而在 **properties** 文件中则无法配置，必须通过代码中的 **Hard Coding** 加载相应的映射文件。下面如果不做特别说明，都是指基于 XML 格式文件的配置方式。

Hibernate 配置文件名默认为“**hibernate.cfg.xml**”（或者 **hibernate.properties**），Hibernate 初始化期间会自动在 **CLASSPATH** 中寻找这个文件，并读取其中的配置信息，为后期数据库操作做好准备。

配置文件应部署在 **CLASSPATH** 中，对于 Web 应用而言，配置文件应放置在 **WEB-INF\classes** 目录下。

（1）hibernate.cfg.xml 配置文件。

【例 5-22】 一个典型的 **hibernate.cfg.xml** 配置文件。

```
< ? xml version = '1.0' encoding = 'utf -8' ? >
<! DOCTYPE hibernate -configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

```

<hibernate - configuration>
<session-factory>
<!-- -JDBC 驱动程序- - >
<property name= "connection.driver_class">com.mysql.jdbc.Driver
    </property>
<!-- -JDBC URL- - >
<property name= "connection.url">jdbc:mysql://localhost:3306/hdata1
    </property>
<!-- -数据库使用者- - >
<property name= "connection.username">root</property>
<!-- -数据库密码- - >
<property name= "connection.password">root</property>
<!-- -SQL 方言，这边设定的是 MySQL- - >
<property name= "dialect">org.hibernate.dialect.MySQLDialect
    </property>
<!-- -显示实际操作数据库时的 SQL- - >
<property name= "show_sql">true</property>
<!-- -事务管理类型，这里我们使用 JDBC Transaction- - >
<property name= "hibernate.transaction.factory_class">
Org.hibernate.transaction.JDBCTransactionFactory
</property>
<!-- -映射文件配置，注意配置文件名必须包含其相对于根的全路径- - >
<mapping resource = "liufy/test/Person.hbm.xml"/>
<mapping resource = "liufy/test/User.hbm.xml"/>
</session-factory>

```

(2) hibernate.properties 配置文件。

【例 5-23】 一个典型的 hibernate.properties 配置文件。

```

hibernate.dialect org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class com.mysql.jdbc.Driver
hibernate.connection.url jdbc:mysql://localhost:3306/hdata1
hibernate.connection.username root
hibernate.connection.password root

```

在下面的例 5-24 FirstHibernate.java 中，可以看到，程序中通过少量代码实现了 Java 对象和数据库数据的同步，同时借助 Hibernate 的有力支持，轻松实现了对象到关系型数据库的映射。相对于传统的 JDBC 数据访问模式，这样的实现无疑更符合面向对象的思想，同时也大大提高了开发效率。

构建 Hibernate 基础代码通常有以下 3 种途径。

① 手工编写。

② 直接从数据库中导出表结构，并生成对应的 ORM 文件和 Java 代码。这是实际开发中最常用的方式，也是这里所推荐的方式。通过直接从目标数据库中导出数据结构，最小化了手工编码和调整的可能性，从而在最大程度上保证了 ORM 文件和 Java 代码与实际数据库结构相一致，如采用 Synchronizer、Middlegen 等工具。

③ 根据现有的 Java 代码生成对应的映射文件，将 Java 代码与数据库表进行绑定。通过预先编写好的 POJO 生成映射文件，这种方式在实际开发中也经常使用，特别是结合了

XDoclet 之后显得尤为灵活，其潜在问题就是与实际数据库结构之间可能出现同步上的障碍。由于需要手工调整代码，往往在调整的过程中由于手工操作的疏漏，导致最后生成的配置文件错误，这点需要在开发中特别注意。

4) 运行实例

下面编写一个测试的程序 FirstHibernate.java。这个程序直接以 Java 程序设计人员熟悉的语法方式来操作对象，而实际上也直接完成对数据库的操作，程序会将一条记录数据存入表格中。

【例 5-24】 程序 FirstHibernate.java 将一条记录数据存入表中。

```
//FirstHibernate.java
package liufy;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
public class FirstHibernate {
public static void main(String [] args) {
    //Configuration 负责管理 Hibernate 配置信息
    Configuration config = new Configuration().configure();
    //根据 config 建立 SessionFactory
    //SessionFactory 将用于建立 Session
    SessionFactory sessionFactory = config.buildSessionFactory();
    //将持久化的物件
    User user = new User();
    user.setName("JavaBoy");
    user.setAge(new Integer(40));
    //开启 Session，相当于开启 JDBC 的 Connection
    Session session = sessionFactory.openSession();
    //Transaction 表示一组对 DB 的交易
    Transaction tx = session.beginTransaction ();
    //将对象映像至数据库表中储存
    session.save(user);
    tx.commit();
    session.close();
    sessionFactory.close();
    System.out.println("新增记录成功，请在 MySQL 中观看结果！");
}
}
```

编译及运行：

```
D:\chap05\hbexample1>javac -d .FirstHibernate.java
```

```
D:\chap05\hbexample1>java liufy.FirstHibernate
```

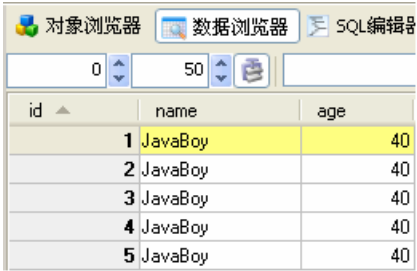
运行结果：

```
FirstHibernate: insert into user <name, age> values <?,?>
```

新增记录成功，请在 MySQL 中观看结果！

在本程序中，Configuration 代表了 Java 对象至数据库的映像设定，这个设定是从上面

的 XML 而来的，接下来从 Configuration 取得 SessionFactory 对象，并由它来开启一个 Session，它代表对象与表的一次会话操作，而 Transaction 则表示一组会话操作，只需要直接操作 User 对象，并进行 Session 与 Transaction 的相关操作，Hibernate 就会自动完成对数据库的操作。多次运行上面的程序，在 MySQL 中可以得到如图 5-17 所示的结果。



The screenshot shows a window titled '对象浏览器' (Object Browser) with a sub-tab '数据浏览器' (Data Browser). It displays a table with columns 'id', 'name', and 'age'. The table contains 5 rows of data, all with 'name' as 'JavaBoy' and 'age' as 40. The first row is highlighted in yellow.

id	name	age
1	JavaBoy	40
2	JavaBoy	40
3	JavaBoy	40
4	JavaBoy	40
5	JavaBoy	40

图 5-17 数据库中的结果

接下来介绍使用 Hibernate 进行数据库的简单查询，文件为 SecondHibernate.java。
当储存数据之后，更重要的是如何将记录读出。Hibernate 也可以让用户不写 SQL 语句，而以 Java 中操作对象的习惯来查询数据。

【例 5-25】程序 SecondHibernate.java 使用 Hibernate 进行数据库的简单查询。

```
//SecondHibernate.java
package liufy;
import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.criterion.Expression;
import java.util.Iterator;
import java.util.List;
public class SecondHibernate {
public static void main (String[] args) {
    Configuration config = new Configuration ().configure();
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();
    Criteria criteria = session.createCriteria(User.class);
    //查询 user 所有字段
    List users = criteria.list ();
    Iterator iterator = users.iterator();
    System.out.println("id\tname\tage");
    while (iterator.hasNext()) {
        User user = (User) iterator.next();
        System.out.println(user.getId()+"\t"+user.getName()+"\t"+user.getAge());
    }
    //查询 user 中符合条件的字段
    criteria.add(Expression.eq("name", "JavaBoy"));
    users = criteria.list ();
```

```

        iterator = users.iterator ();
        System.out.println("id \tname\tage");
        while ( iterator.hasNext () ) {
            User user = (User) iterator.next();
            System.out.println(user.getId() + "\t" + user.getName() + "\t" +
                user.getAge () );
        }
        session.close ();
        sessionFactory.close ();
    }
}

```

Criteria 对 SQL 进行封装，对于不了解 SQL 语言的开发人员来说，使用 Criteria 也可以轻易地进行各种数据的检索。用户可以使用 Expression 设定查询条件，并将之加入 Criteria 中对查询结果进行限制，Expression.eq () 表示设定符合条件的查询。例如，Expression.eq ("name", "JavaBoy") 表示设定查询条件为 “name” 字段中为 “JavaBoy” 的数据。

执行结果如下：

```

Hibernate:select this_.id as id0_0_,this_.name as name0_0_,this_.age as
age0_0_ from user this_

```

id	name	age
1	JavaBoy	25

```

Hibernate:select this_.id as id0_0_,this_.name as name0_0_,this_.age as
age0_0_ from user this_ where this_.name = ?

```

id	name	age
1	JavaBoy	25

上例也可以通过 HQL (Hibernate Query Language) 来进行查询。关于 Criteria 条件查询语句及 HQL 见第 5.3.4 节。

2. Hibernate 的主要接口

例 5-25 的代码中引入了几个 Hibernate 基础语义：Configuration、SessionFactory 和 Session。下面就这几个关键概念进行探讨。

1) Configuration

Configuration 类负责管理 Hibernate 的配置信息。Hibernate 运行时需要获取一些底层实现的基本信息，其中几个关键属性包括：数据库 URL、数据库用户、数据库用户密码、数据库 JDBC 驱动类。

数据库 dialect，用于对特定数据库提供支持，其中包含了针对特定数据特性的实现，如 Hibernate 数据类型到特定数据库数据类型的映射等。

使用 Hibernate 必须首先提供这些基础信息以完成初始化工作，为后继操作做好准备。这些属性在 Hibernate 配置文件 (hibernate.cfg.xml 或 hibernate.properties) 中加以设定。当调用：

```

Configuration config = new Configuration().configure();

```

时，Hibernate 会自动在当前的 CLASSPATH 中搜寻 hibernate.cfg.xml 文件，并将其读取到内存中作为后继操作的基础配置。Configuration 类一般只有在获取 SessionFactory 时需要涉

及。当获取 `SessionFactory` 之后，由于配置信息已经由 `Hibernate` 维护并绑定在返回的 `SessionFactory` 之上，因此，一般情况下无需再对其进行操作。

如果不希望使用默认的 `hibernate.cfg.xml` 文件作为配置文件的话，可以使用如下代码：

```
File file = new File("c:\\sample\\myhibernate.xml");
Configuration config = new Configuration().configure(file);
```

2) SessionFactory

`SessionFactory` 负责创建 `Session` 实例，可以通过 `Configuration` 实例构建 `SessionFactory`：

```
Configuration config = new Configuration().configure();
SessionFactory sessionFactory = config.buildSessionFactory();
```

`Configuration` 实例 `config` 会根据当前的配置信息，构造 `SessionFactory` 实例并返回。

`SessionFactory` 一旦构造完毕，即被赋予特定的配置信息。也就是说，之后 `config` 的任何变动将不会影响到已经创建的 `SessionFactory` 实例。如果需要使用基于改动后的 `config` 实例的 `SessionFactory`，需要从 `config` 重新构建一个 `SessionFactory` 实例。

3) Session

`Session` 是持久层操作的基础，相当于 `JDBC` 中的 `Connection`。`Session` 实例通过 `SessionFactory` 实例构建：

```
Configuration config = new Configuration().configure();
SessionFactory sessionFactory = config.buildSessionFactory();
Session session = sessionFactory.openSession();
```

之后就可以调用 `Session` 所提供的 `save`、`find` 和 `flush` 等方法完成持久层操作。

5.3.3 DAO模式、Hibernate Synchronizer插件及开发

1. DAO模式

`DAO` (`Data Access Object`) 模式是设计关系数据库系统结构的对象类的集合。它们提供了完成管理一个关系型数据库系统所需的全部操作的属性和方法，例如，创建数据库；定义表、字段和索引；建立表间的关系；定位和查询数据库等，使得编写 `Hibernate` 的配置文件更容易和简单。

数据源不同，数据访问也不同。根据存储的类型（关系数据库、面向对象数据库、文件等）和供应商实现不同，持久性存储（如数据库）的访问差别也很大。

例如，在一个应用系统中使用 `JDBC API` 对 `Oracle` 数据库进行连接和数据访问，这些 `JDBC API` 与 `SQL` 语句分散在系统各个程序文件中，当需要其他 `RDBMS`（如 `INFORMIX`）时，就需要重写数据库连接和访问数据的模块。

一个软件模块（类、函数、代码块等）在扩展性方面应该是开放的，而在更改性方面应该是封闭的，这就是开闭原则。要实现这个原则，在软件面向对象设计时要考虑接口封装机制、抽象机制和多态技术。这里的关键是将软件模块的功能部分和不同的实现细节清晰地分开。在数据库访问对象中，应该运用这个原则。

在数据库编程的时候，经常遇到这种情况，一个用户的数据访问对象里的操作方法有 `insert`、`delete`、`update`、`select` 等，对不同数据库其实现的细节是不同的。因此，不太可能针对每种类型的数据库做一个通用的对象来实现这些操作。但是，可以定义一个用户数据访问对象的接口 `IUserDAO`，提供 `insert`、`delete`、`update`、`select` 等抽象方法。不同类型数据库的用户访问对象实现这个接口就可以了，如图 5-18 所示。

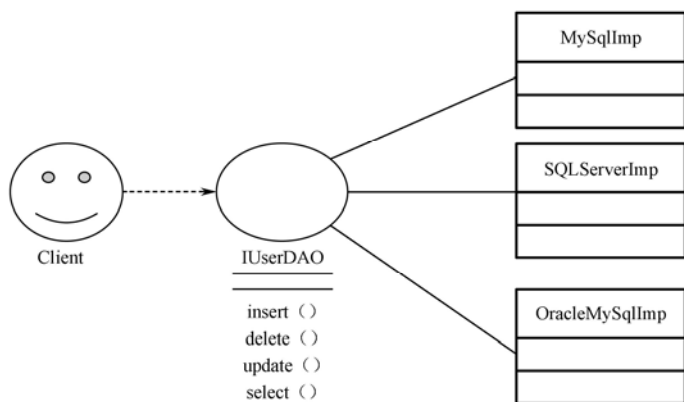


图 5-18 DAO

DAO 是 Data Access Object 数据访问对象（接口），它介于数据库资源和业务逻辑之间，其意图是将底层数据访问操作与高层业务逻辑完全分开。

2. Hibernate Synchronizer 插件简介

Hibernate Synchronizer 是一个 Eclipse 插件，可以自动生成*.hbm.xml 文件、持久化类和 DAO 模式。

Hibernate Synchronizer 支持 db-hbm-pojo-dao 的自动同步更改，支持 Eclipse2 和 3 系列的版本。如果用户之前采用 Middlegen 来进行 DB Schema-hbm 转换，会发现在手工更改 hbm 中的某些特性，再使用 Middlegen 来同步 DB Schema-hbm 时，手工更改的信息将会丢失；HibernateSynchronizer 则不会使之丢失。

Hibernate Synchronizer 插件在修改映射文档时自动更新 Java 代码。通过为每个被映射的对象创建一对类，它比 Hibernate 的内置代码生成工具更先进。它“拥有”一个基类，当用户修改映射时，它可以随意重写这个基类。它还提供一个扩展这个基类的子类，可以在这个子类中加入业务逻辑和其他代码。

Hibernate Synchronizer 还提供了用于 Eclipse 的新编辑器组件，为此，类文档提供智能辅助和代码自动完成功能。该编辑器提供了一个映射中的属性和关系的图形化视图、创建新元素的“向导”界面。而且，在其默认配置中，编辑器会在用户编辑映射文档时自动重新生成数据访问类。

Hibernate Synchronizer 还有其他的功能。例如，它在 Eclipse 的“New”菜单中加入了一个区域，为创建 Hibernate 配置和映射文件提供向导；并在包的资源管理器和其他适当的位置中添加了上下文菜单项，使用户可以轻松访问相关的 Hibernate 操作。要了解 Hibernate Synchronizer 插件的详细信息可以访问：<http://hibernatesync.sourceforge.net/>。

Hibernate Synchronizer 的主要功能包括以下几方面。

- 通过一个向导配置并生成 Hibernate Configuration File。
- 通过一个向导同步生成数据库表的*.hbm.xml 文件。
- 通过*.hbm.xml 文件同步生成 Hibernate 持久化类和 DAO。
- 提供 Hibernate Synchronizer editor 编辑*.hbm.xml 文件。
- 用一种叫做 Velocity 的语言定制个性化的代码和资源生成模板。

3. Hibernate Synchronizer的获取与安装

Hibernate Synchronizer 插件可以通过传统的方法方式来进行安装，其官方网站的下载地址是：<http://www.binamics.com/hibernatesync>。也可以从其他网站上下载 Hibernate Synchronizer 3.1.9。下载以后，把文件复制到 Eclipse 的 plugins 目录和 feature 目录中。下载完成后，重启 Eclipse 即安装成功。

4. Eclipse中使用Hibernate Synchronizer进行开发

本节将介绍如何使用 Hibernate Synchronizer 插件进行开发。参照前面 5.3.2 节的 User 表例子，再创建一个数据表 Person。手动编写配置文件是一件较麻烦的事情，而且也很容易出错。本节将使用 Hibernate Synchronizer 插件使这一切变得很简单。下面演示一个最简单的单表操作，以便熟悉使用 Hibernate Synchronizer 的开发过程。

1) 在项目中使用 Hibernate

(1) 先进行数据库表的准备。选择 MySQL 数据来设计这个应用，首先在 MySQL 里建立一个新的数据库为 Hdata1，再建立一个数据表，名为 Person，包含 ID、Name、Sex、Address 四个字段，建表的 SQL 语句如下：

```
CREATE TABLE 'person' (  
    'ID' int(11) NOT NULL auto_increment,  
    'Name' varchar(20) collate gb2312_bin NOT NULL default '',  
    'Sex' char(1) collate gb2312_bin default NULL,  
    'Address' varchar(200) collate gb2312_bin default NULL,  
    PRIMARY KEY('ID')  
) ENGINE = MyISAM;
```

(2) 新建一个普通的 Java 项目。单击“File”→“Project”→“New Project”→“Java Project”，输入项目名称：HibernateTest，如图 5-19 所示。

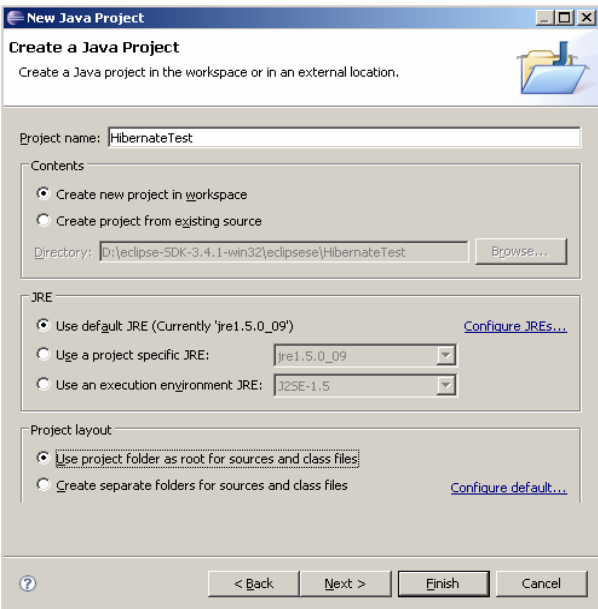


图 5-19 输入工程名称

注意：加入 Hibernate 的所有 lib 文件，包括 Hibernate 下面的 hibernate3.jar 和 lib 目录下面的所有.jar 文件，还要加入 MySQL JDBC 驱动文件，如 mysql-connector-java-3.0.14-production-bin.jar（驱动程序自己选择加载，版本不同，文件名也不同），如图 5-20 所示。也可以采用加入 User Library 的方式将加入的 JAR 文件放入一个文件夹（如 lib）中。接着建立 src 目录，如图 5-21 所示。

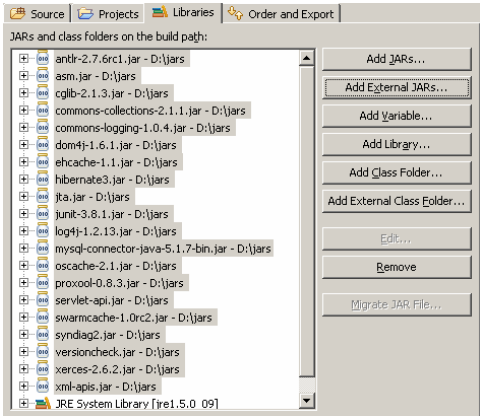


图 5-20 导入涉及的包

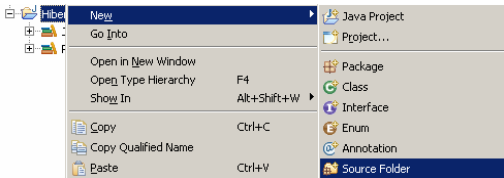


图 5-21 创建 src 目录

2) 创建 Hibernate Configuration File 文件

在项目中加入一个 Hibernate 的配置文件，然后在 src 目录下选择“New”→“Other”→“Hibernate”→“Hibernate Configuration File”。

在弹出的界面中，用户需要指定要使用的数据库，以及连接数据库所需要的信息，对应地选择数据库为 MySQL，并配置了数据库的 URL 和管理员账号与密码，如图 5-22 所示。

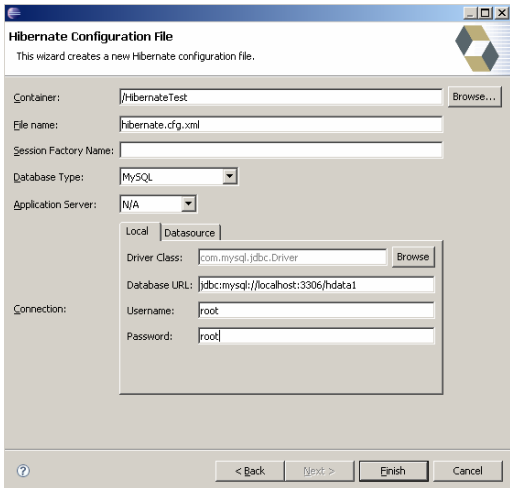


图 5-22 配置 Hibernate 图形界面

单击“Browse”按钮，在弹出的对话框中输入“Driver”，在下面就会出现相应的驱动所在的包。选中 com.mysql.jdbc.Driver 所在的包的文件，单击“OK”按钮即可，如图 5-23 所示。

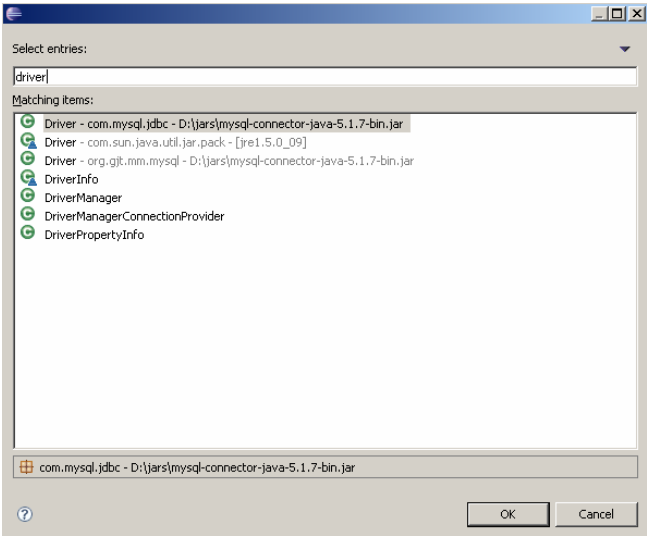


图 5-23 配置数据库驱动程序

在图 5-24 中，也可以指定从 JNDI 数据源中获得连接，单击“Datasource”选项卡进行配置。

单击“Finish”按钮之后，系统会自己生成一个名为 hibernate.cfg.xml 的文件，里面包含了基本的配置信息。如果需要高级配置，可以手动配置，也可以通过其他插件来进行编辑，如 MyEclipse 的 XML Editor。

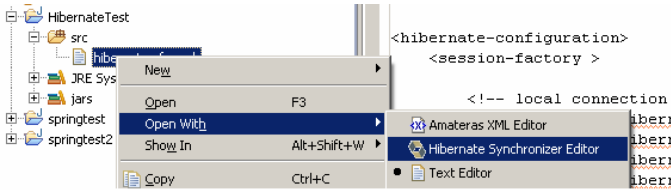


图 5-24 修改 Hibernate 配置文件

3) 创建映射文件

下面要生成映射文件，首先新建一个包，单击“New”→“Package”，如图 5-25 所示。在打开的对话框中，输入“liufy.test”。

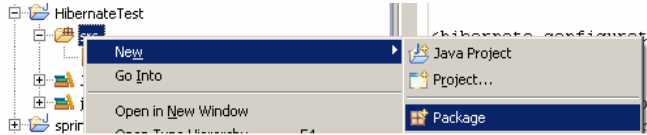


图 5-25 新建 liufy.test 包

在 liufy.test 包下选择“New”→“Other”→“Hibernate”→“Hibernate Mapping File”，在弹出的界面中单击“Refresh”按钮，将会列出库中所有的数据表。选中我们要使用的 Person 表，单击“Browse”按钮，选择所要生成的 POJO 文件所在的包：liufy.test，如图 5-26 所示。

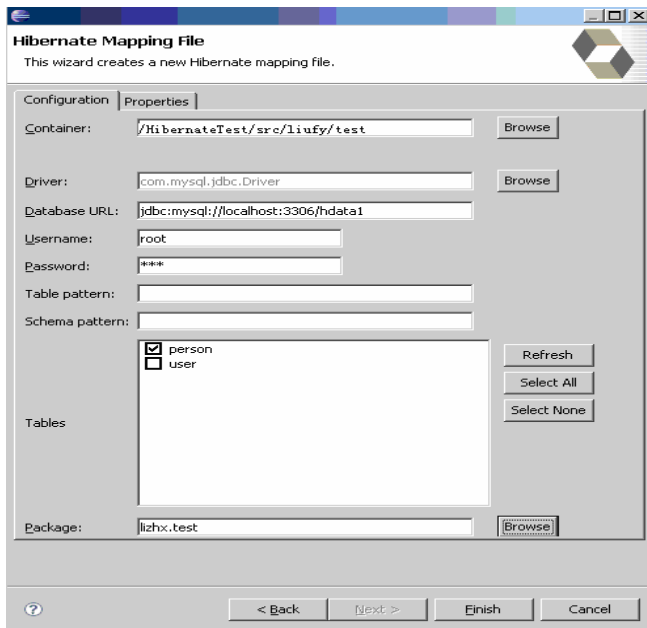


图 5-26 配置 POJO 文件选择数据库表

在上述界面的“Properties”选项卡中可以配置 hbm.xml 的其他选项，包括文件扩展名、聚合列名，ID 生成规则等。设置完成后，系统会自动生成一个名为 Person.hbm.xml 的文件，可以通过这个文件生成相关的存根类，如图 5-27 所示。

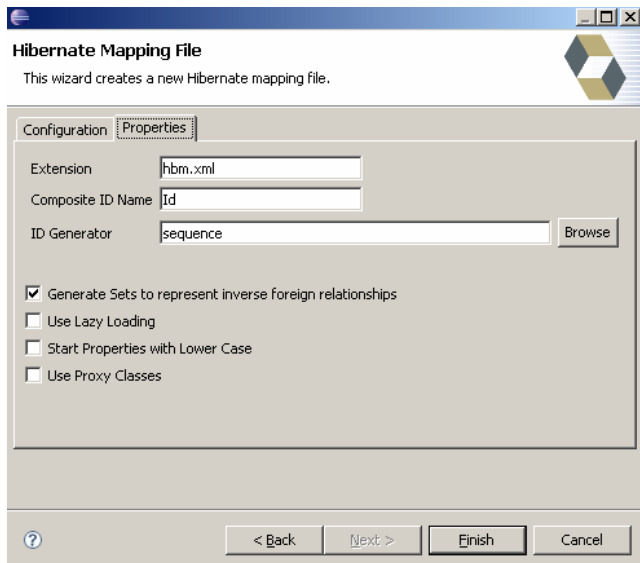


图 5-27 配置 POJO 的参数图形界面

在 Person.hbm.xml 文件上单击鼠标右键，在弹出的菜单中选择“Hibernate Synchronizer”→“Synchronize Files”，如图 5-28 所示。该操作成功后生成 Person.hbm.xml，编辑 Person.hbm.xml，将下面语句中 false 改成 true，如图 5-29 所示，然后存盘。

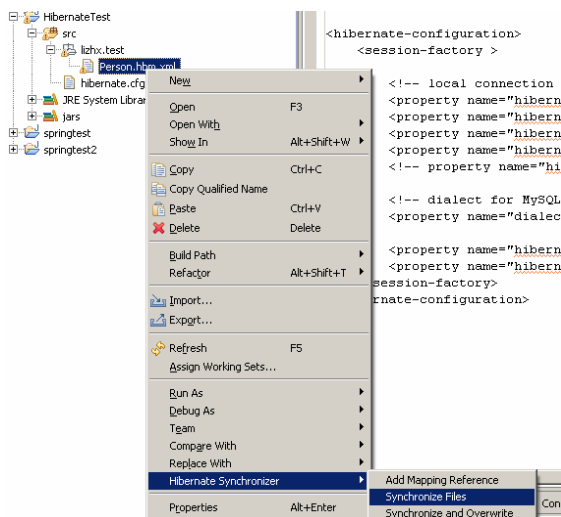


图 5-28 生成 POJO 的图形界面

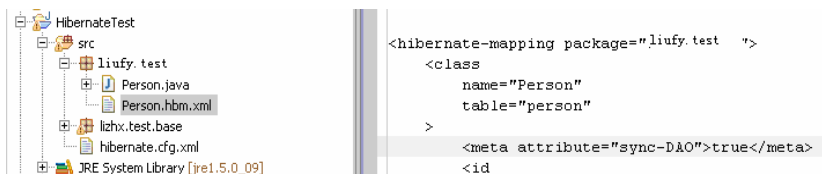


图 5-29 修改映射文件

该操作将生成 2 个包，8 个类文件，里面封装了 Hibernate 的操作细节，让程序员可以专心面对业务逻辑的开发。

base 包中存放插件生成的 5 个抽象类，在 Hibernate Synchronizer “再同步” 时会覆盖 base 包中的类，因此，用户不要把客户代码放在 base 包中的类里。换句话说，任何时候都不要修改这些类。

dao 包中存放了 3 个抽象类，分别继承自 base 包中相应的三个类。dao 包中的三个类完全是空的实现，客户可以在这里插入自己的代码。采用这样的结构，就把客户代码从插件生成的代码中分离出来，既实现了客户对插件生成代码的定制，又不会在插件 “再同步” 时影响到客户代码。

仔细阅读这些文件可以进一步提高对 Hibernate 的认识，增长应用技巧。

如果没有将 false 改成 true，那么，系统将只生成 1 个包，5 个类文件，不包括 DAO 模式。这时读者也可以编写自己的 DAO。

接下来，需要在 Hibernate 的配置文件添加 Person 的相关信息。在 Person.hbm.xml 上单击鼠标右键，在弹出的菜单中，选择 “Synchronizer” → “Add Mapping Reference”。操作成功后，在 hibernate.cfg.xml 中即加入了 Person 的映射文件，如图 5-30 所示。

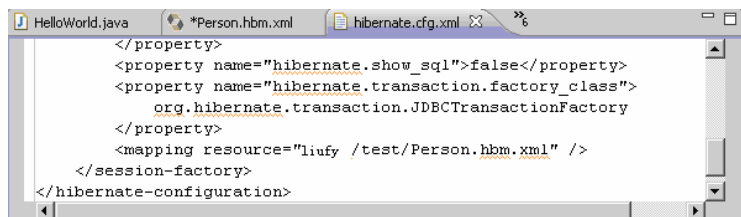


图 5-30 Hibernate 的配置文件

经过 DAO 模式封装后的 PersonDAO 接口内容如下：

【例 5-26】 DAO 模式封装后的 PersonDAO 接口内容。

```
package liufy.test.dao.iface;

public interface PersonDAO {

    public liufy.test.Person get(java.lang.Integer key);
    public liufy.test.Person load(java.lang.Integer key);
    public java.util.List findAll ();
    public java.lang.Integer save (liufy.test.Person person);
    public void saveOrUpdate (liufy.test.Person person);
    public void update (liufy.test.Person person);
    public void delete (java.lang.Integer id);
    public void delete (liufy.test.Person person);
}
```

上面是基本的 CRUD 操作：

- save () 方法：把 Java 对象保存数据库中。
- update () 方法：更新数据库中的 Java 对象。
- delete ()方法：把 Java 对象从数据库中删除。
- load ()方法：从数据库中加载 Java 对象。
- find ()方法：从数据库中查询 Java 对象。

更详细的封装可以查阅 BasePersonDAO.java。

4) 运行 Hibernate 实例

在 src 中建立包 liufy.app，在该包中建立类 Test.java，如图 5-31 所示。该程序的作用是在数据库里增加一条新的记录。这个类的代码不会被插件修改。

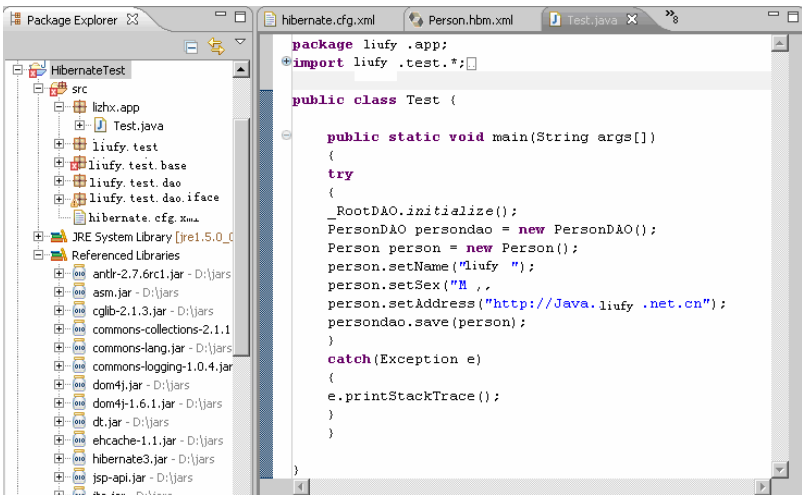


图 5-31 编辑 Test.java

【例 5-27】 Test.java 的内容。

```
package liufy.app;

import liufy.test.* ;
import liufy.test.dao.* ;

public class Test {

    public static void main (String args[] ) {
```

```

try {
    _RootDAO.initialize ();
    PersonDAO persondao = new PersonDAO();
    Person person = new Person ();
    person.setName ("Liufy");
    person.setSex ("M");
    person.setAddress("http://Java.liufy.net.cn");
    persondao.save (person);
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

代码中 `_RootDAO.initialize ()` 是必须的。Hibernate Synchronizer 生成的持久对象是标准的 Hibernate 持久对象，包含一组 set 和 get 方法。DAOs 负责操作持久对象，包括对 session 和事务管理、load 和释放对象、保存或更新、查询等功能。

可以看出，插件已经把 session 操作和事务操作都封装起来了，代码工作得到了极大的简化，而且可以利用插件自带的 Hibernate Editor 来编辑 hbm.xml 文件，非常方便。

此时，还需要把 ID 的生成方式改为 identity，用鼠标右键单击 `Person.hbm.xml`，选择“Open With”→“Hibernate Synchronizer Editor”，把 ID 的生成方式改为 identity。

要让这个程序正常运行，还需要对配置文件 `hibernate.cfg.xml` 做一些修改。

使用 Eclipse 的文本编辑器打开该文件，其中有如下的内容：

```

<!--
<property name = "hibernate.transaction.factory_class">
    org.hibernate.transaction.JDBCTransactionFactory
</property>
-->

```

由于在例子中，并没有使用 JTA 来控制事务，所以应注销上面的内容，`Test.java` 程序才能正常运行。

以上各项设置完成后即可运行程序。选择 `Test.java`，单击“Run”按钮，在出现的配置中选择 Java Application，如图 5-32 所示。

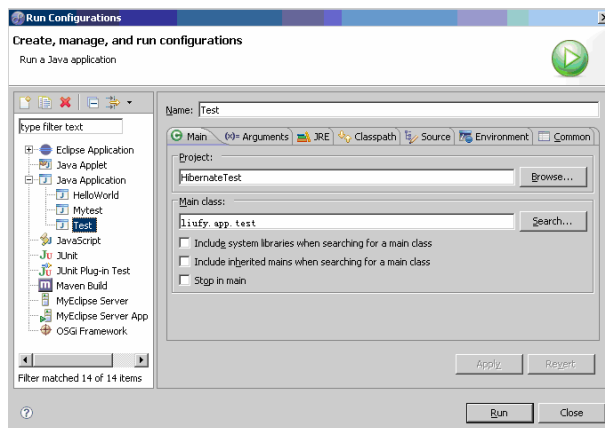
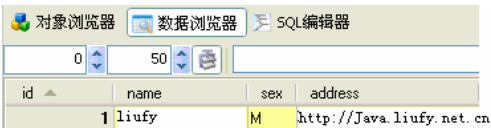


图 5-32 运行 Test.java

单击“Run”按钮开始运行，如果以上各步操作正确的话，可以看到数据已经被保存到数据库中，如图 5-33 所示。



id	name	sex	address
1	liufy	M	http://Java.liufy.net.cn

图 5-33 数据库内 Person 表的内容

如果在实际开发工作中，需要重新设计数据表结构，那么只需要在.hbm.xml 文件中做相应的修改，然后执行 Synchronize and Overwrite 的操作，插件会重新生成存根文件，只需要修改程序逻辑即可，非常方便。有了这样的功能插件，可以从配置文件的编写、查错中解脱出来，从而提高工作效率。

对 Test.java 程序稍加修改，应用 get、load、findAll、save、saveOrUpdate、update 和 delete 等方法，即可完成对对象的删除、更新、查询全部对象等操作。该部分作为练习，请读者自行完成。

虽然 get 和 load 方法都可以取得 POJO 对象实体，但还是有区别的。

- (1) get()在类检索级别时总是执行立即检索而且如果检索不到相关的对象的话会返回 null，load()方法则会抛出一个 ObjectNotFoundException。
- (2) load()方法可返回一个实体代理类类型，而 get() 方法直接返回的是实体类对象。
- (3) load()方法可以充分利用内部缓存和二级缓存，而 get() 方法会忽略二级缓存，若内部缓存没有查询到结果，则会到数据库中去查询。

5.3.4 Criteria Query、HQL数据查询语言及Query接口

1. 介绍Criteria Query和HQL

在 Hibernate 中，数据查询与检索机制很完善。相对其他 ORM 实现而言，Hibernate 提供了灵活多样的查询机制。其中包括：Criteria Query（条件查询 API）、Hibernate Query Language（HQL）、SQL。

下面分别讲述 Criteria Query 和 HQL。在 Hibernate 中不提倡使用 SQL。

1) Criteria Query

Criteria Query 通过面向对象化的设计，将数据查询条件封装为一个对象。简单来讲，Criteria Query 可以看做传统 SQL 的对象化表示。

【例 5-28】Criteria 实例的封装。

```
criteria criteria = session.createCriteria(Person.class);
criteria.add(Expression.eq("name","Liufy"));
criteria.add(Expression.eq("sex","M"));
```

这里的 Criteria 实例实际上是 SQL “Select *form Person where name = 'Liufy' and sex = 'M'”的封装（可以打开 Hibernate 的 show_sql 选项，以观察 Hibernate 在运行期生成的 SQL 语句）。Hibernate 在运行期间会根据 Criteria 中指定的查询条件（也就是上面代码中通过 criteria.add 方法添加的查询表达式）生成相应的 SQL 语句。

这种方式的特点是比较符合 Java 程序员的编码习惯，并且具备清晰的可读性。正因为此，不少 ORM 实现中都提供了类似的实现机制（如 Apache OJB）。对于 Hibernate 的初学

者，特别是对 SQL 了解有限的程序员而言，Criteria Query 无疑是上手的极佳途径。相对于 HQL，Criteria Query 提供了更易于理解的查询手段，借助 IDE 的 Coding Assist 机制，Criteria 的使用能够在较短的时间内掌握。

Criteria 本身只是一个查询容器，具体的查询条件需要通过 Criteria.add 方法添加到 Criteria 实例中。如例 5-28 所示，Expression 对象具体描述了查询条件。针对 SQL 语法，Expression 提供了对应的查询限定机制，见表 5-4。

表 5-4 Criteria 查询表达式

方 法	描 述
Expression.eq	对应 SQL “field = value”表达式，如 Expression.eq(“name”, “Liufy”)
Expression.allEq	参数为一个 Map 对象，其中包含了多个属性值对应关系。相应于多个 Expression.eq 关系的叠加
Expression.gt	对应 SQL 中的 “field>value”表达式
Expression.ge	对应 SQL 中的 “field>=value”表达式
Expression.lt	对应 SQL 中的 “field<value”表达式
Expression.le	对应 SQL 中的 “field<=value”表达式
Expression.between	对应 SQL 中的 “between”表达式，如表示年龄（age）位于 23~40 之间，即 Expression.between(“age”,new Integer(23),new Integer(40));
Expression.like	对应 SQL 中的 “field like value”表达式
Experssion.in	对应 SQL 中的 “field in...”表达式
Expression.eqProperty	用于比较两个属性之间的值，对应 SQL 中的 “field=field”，例如，Expression.eqProperty (“Persion.ID”, “Class.CID”);
Expression.gtProperty	用于比较两个属性之间的值，对应 SQL 中的 “field>field”
Expression.geProperty	用于比较两个属性之间的值，对应 SQL 中的 “field>=field”
Expression.ltProperty	用于比较两个属性之间的值，对应 SQL 中的 “field<field”
Expression.leProperty	用于比较两个属性之间的值，对应 SQL 中的 “field<=field”
Expression.and	and 关系组合，如 Expression.and(Expression.eq(“name”, “Liufy”),Expression.eq(“sex”, “M”));
Expression.or	or 关系组合
Expression.sql	作为补充，本方法提供了原生 SQL 语法的支持。下面的代码返回所有名称以 “Liufy” 起始的记录： Expression.sql(“lower({alias}.name)like lower(?)”, “Liufy %”,Hibernate.STRING);其中的 “{alias}”将由 Hibernate 在运行期间使用当前关联的 POJO 别名替换

注意，这里所谓的属性名是 POJO 中对应表字段的属性名（大小写敏感），而非库表中的实际字段名称。

另外，Criteria 还有一些高级特性。

（1）限定返回的记录范围。通过 criteria.setFirstResult/setMaxResults 方法可以限制一次查询返回的记录范围：

```
criteria criteria = session.createCriteria(Person.class);
//限定查询返回检索结果中，从第一百条结果开始的 20 条记录
criteria.setFirstResult(100);
criteria.setMaxResults(20);
```

（2）对查询结果进行排序。

【例 5-29】对查询结果进行排序。

```
//查询所有 id = 2 的记录
//并分别按照 name (顺序)和 sex (逆序)排序
```

```
criteria criteria = session.createCriteria(Person.class);
criteria.add(Expression.eq( "id",new Integer(2) ) );
criteria.addOrder(Order.asc("name") );
criteria.addOrder(Order.desc("sex") );
```

(3) 限制结果集内容。

【例 5-30】限制结果集内容。

```
List persons = session.createCriteria(Person.class)
.add(Restrictions.like("name", "Liufy %") )
.add(Restrictions.between("weight",min Weight,maxWeight) )
.list();
//Restrictions 类是 Criterion 类型的父类
```

(4) 结果集排序。

【例 5-31】结果集排序。

```
List persons = sess.createCriteria(Person.class)
.add(Restrictions.like("name", "L%")
.addOrder(Order.asc("name") ).addOrder(Order.desc("sex") )
.setMaxResults(50)
.list();
```

(5) 关联。使用 createCriteria()可以非常容易地在互相关联的实体间建立约束。

【例 5-32】关联。

```
List persons = session.createCriteria(Person.class)
.add(Restrictiions.like("name", "F%")
.createCriteria("kittens")
.add(Restrictions.like("name", "F%")
.list();
```

注意：第二个 createCriteria()返回一个新的 Criteria 实例。该实例引用 kittens 集合中的元素。接下来，替换形态在某些情况下也是很有用的。

```
List persons = sess.createCriteria(Person.class)
.createAlias("Mits", "kt");//createAlias()并不创建一个新的 Criteria 实例
.createAlias("Mate", "mt")
.add(Restrictions.eqProperty("kt.name", "mt.name") )
.list();
```

其他的高级特性，如动态关联抓取，通过示例构建查询、投影、聚合、分组，以及离线查询等，读者可以参考 Hibernate 的官方网站。

由于 Hibernate 在实现过程中将精力更加集中在 HQL 查询语言上，因此，Criteria 的功能实现还没做到尽善尽美。在实际开发中，建议还是采用 Hibernate 官方推荐的查询封装模式：HQL。

2) Hibernate Query Language (HQL)

传统的 SQL 语言采用的是结构化的查询方法，而这种方法对于查询以对象形式存在的数据却无能为力。幸运地是，Hibernate 为用户提供一种语法类似于 SQL 的语言，Hibernate 查询语言 (HQL)。和 SQL 不同的是，HQL 是一种面向对象的查询语言，它可以查询以对象形式存在的数据。

相对 Criteria，HQL 提供了更接近传统 SQL 语句的查询语法，也提供了更全面的特性。

【例 5-33】 HQL 最简单的一个例子。

```
String hql = "from Liufy.test.Person as p";  
Query query = session.createQuery(hql);  
List userList = query.list();
```

上面的代码将取得 **Person** 的所有对应记录。

如果需要取出名为“Liufy”的用户的记录，可以通过 SQL 语句加以限定：

```
String hql = "from Liufy.test.Person as pers where  
            Pers.Name = 'Liufy' ";  
Query query = session.createQuery(hql);  
List persList = query.list();
```

其中，新引入了两个子句“as”和“where”，as 子句为类名创建了一个别名（as 可以省略），而 where 子句指定了限定条件。

注意：HQL 子句本身大小写不敏感，但是其中出现的类名和属性名必须注意区分大小写。

SQL 本身是非常强大的。当 SQL 的这种强大和处理面向对象数据的能力相结合时，就产生了 HQL。与 SQL 一样，HQL 提供了丰富的查询功能，如投影查询、聚合函数、分组和约束。任何复杂的 SQL 都可以映射成 HQL。

一个 ORM 框架是建立在面向对象的基础上的。最好的例子是 **Hibernate** 如何提供类 SQL 查询。虽然 HQL 的语法类似于 SQL，但实际上它的查询目标是对象。HQL 拥有面向对象语言的所有的特性，这其中包括多态、继承和组合。这就相当于一个面向对象的 SQL。为了提供更强大的功能，HQL 还提供了很多的查询函数。这些函数可以被分为 4 类：投影函数、约束函数、聚合函数和分组函数。下面将分别简单介绍 HQL 的这 4 类子函数。

（1）投影函数：投影就是一个可以访问的对象或对象的属性。在 HQL 中，可以使用 from 和 select 子句来完成这项工作。假设 Order 和 Product 分别是订单和产品的 POJO。

from 子句返回指定的类的所有实例，如 from Order 将返回 Order 类的所有实例。换句话说，上面的查询相当于以下的 SQL 语句：

```
select *from order
```

当查询很复杂时，加入别名可以减少语句的长度。例如，下面的 SQL 语句：

```
select o.*,p.*from order o,product p where o.order_id = p.order_id
```

可以很容易地看出，上面的查询是一对多的关系。在 HQL 中相当于一个类中包含多个其他类的实例。因此，以上的 SQL 写成 HQL 就是：

```
from Order as o inner join o.products as product
```

接下来，考虑另外一个从表中得到指定属性的情况。这就是最常用的 select 子句。其在 HQL 中的工作方式和 SQL 中一样。而在 HQL 中，如果只是想得到类的属性的话，select 语句是最好的选择。以上的 SQL 可以使用 select 子句改成如下的 HQL 语句：

```
select product from Order as o inner join o.products as product
```

该 HQL 语句将返回 Order 中的所有 Products 实例。如果要得到对象的某一个属性，可以将 HQL 语句写成如下的形式：

```
select product.name from Order as o inner join o.products as product
```

如果要得到多个对象的属性，可以将 HQL 语句写成如下形式：

```
select o.id,product.name from Order as o inner join o.products as product
```

(2) 约束函数：投影返回的是所有的数据，但在大多数时候并不需要这么多数据，这就需要对数据进行过滤。在 HQL 中，过滤数据的子句与 SQL 中的语句一样，也是 **where**。它的语法类似于 SQL，通过 **where** 子句，可以对其进行过滤。

```
select * from order where id = '1234'
```

这条查询语句中返回了 id 等于 1234 的所有的字段。与这条 SQL 语句对等的是下面的 HQL 语句：

```
select o from Order o where o.id = '1234'
```

从以上两条语句中可以看出，它们的 **where** 子句非常相似。而它们唯一的不同是：SQL 操作的是记录，而 HQL 操作的是对象。

(3) 聚合函数：上述的查询是将每一个记录（对象）当做一个单位，而如果使用聚合，可以将一类记录（对象）当做一个单位。然后再对每一类的记录（对象）进行一系列操作，如对某一列取平均值、求和、统计行数等。HQL 支持以下的聚合函数：**avg**、**sum**、**min**、**max** 和 **count**。

这些聚合函数都返回数值类型，其操作都可以在 **select** 子句中使用：

```
select max(o.priceTotal) + max(p.price) from Order o join o.products p
      group by o.id
```

以上的 HQL 语句返回了两个值的和：**orders** 表中的 **priceTotal** 的最大值和 **products** 表中的 **price** 的最大值之和。还可以使用 **having** 子句对分组进行过滤。如果想按 id 统计 **priceTotal** 小于 1 000 的数量，可按如下的 HQL 语句去实现：

```
select count(o) from Order o having o.priceTotal < 1000 group by o.id
```

还可以将聚合函数和 **having** 子句一起使用。若要按 **products** 表的 id 统计 **price** 小于 **amount** 的平均数的产品数量，HQL 语句如下：

```
select count(p) from Product p having p.price < avg(amount) group by p.id
```

从上面一系列的 HQL 语句中可以看出，所有通过 SQL 实现的功能，都可以通过 HQL 来实现。

(4) 分组函数：分组操作的是行的集合。它根据某一列（属性）对记录集进行分组。这一切是通过 **group** 子句实现的。如下的例子描述了 **group** 子句的一般用法。

```
select count(o) from Order o having o.priceTotal >= 1200 and o.priceTotal
      <= 3200 group by o.id
```

HQL 中的分组和 SQL 中的分组类似。总之，除了一些对 SQL 的特殊扩展外，其他所有的 SQL 功能都可以使用 HQL 描述。

2. 使用Criteria和HQL的简单实例

下面的几个实例是对 Criteria 和 HQL 的应用。本节将使用 5.3.3 节数据库中的设置。读者可以在 5.3.3 节 Eclipse 的工程中直接编写下面的例子。

【例 5-34】使用 Criteria 查询 Person 表中的记录数。

```
//Criteria_test.java
package Liufy.app;
import Liufy.test.*;
import Liufy.test.dao.*;
import org.hibernate.*;
import org.hibernate.criterion.*;
```

```

public class Criteria_test {
    public static void main (String args[ ] ) {
        try {
            _RootDAO.initialize();
            PersonDAO persondao = new PersonDAO();
            Criteria criteria =
(persondao.getSession() ).createCriteria(Person.class);
            criteria.add(Expression.eq( "Name", "Liufy" ) );
            criteria.add(Expression.eq("Sex", "M" ) );
            java.util.List p = criteria.list();
            System.out.println("满足条件的记录数为 " +p.size() ) ";
        }catch(Exception e) {
            e.printStackTrace();
        }
    }
}

```

该程序输出如下:

```

Hibernate:select this_.ID as ID0_0_,this_.Name as Name0_0_,this_.Sex as
Sex0_0_,this
_.Address as Address0_0_from person this_where this_.Name = ? and
this_.Sex = ?

```

满足条件的记录数为 2。

【例 5-35】 使用 HQL 查询 Person 表中的记录数。

```

//HQL_Test.java
package Liufy.app;
import Liufy.test.dao.*;
import org.hibernate.*;
public class HQL_Test {
    public static void main(String args[ ]) {
        try {
            _RootDAO.initialize();
            PersonDAO persondao = new PersonDAO();
            String hql = "from Liufy.test.Person as pers
                        Where pers.Name like 'Liufy %' ";
            Query query = (persondao.getSession()).createQuery(hql);
            java.util.List p = query.list();
            System.out.println("满足条件的记录数为 "+p.size() );
        }catch(Exception e) {
            e.printStackTrace();
        }
    }
}

```

该程序输出如下:

```

Hibernate:select person0_.ID as ID0_,person0_.Name as Name0_,person0_.Sex

```

```
as Sex0_,
person0_.Address as Address0_from person person0_where person0_.Name like
'Liufy %'
满足条件的记录数为 3。
```

3. Hibernate的Query接口

在 Hibernate 2.X 中, find()方法用于执行 HQL 语句, 在 Hibernate 3.0 中废除了 find()方法, 取而代之的是 Query 接口, 它们都用于执行 HQL 语句。实际上, Query 和 HQL 是分不开的。

1) 绑定参数

(1) 使用 “?” 指定参数: 通过 Query 接口可以先设定查询参数, 然后通过 setxxx()等方法, 将指定的参数值填入, 而不用每次都编写完整的 HQL。

【例 5-36】用 “?” 指定参数。

```
Query query=session.createQuery("from Student s where s.age>?and s.name
like?")
```

```
query.setInteger(0,25);           //设置 age > ?中的问号为整型 25
query.setString(1, "%clus%");     //设置 name like?中的问号为字符串"%clus%"
```

(2) 使用 “:” 后加变量的方法设置参数。可以使用命名参数来取代使用 “?” 设置参数的方法, 这可以不用依据特定的顺序来设定参数值。

【例 5-37】用 “:” 后加变量的方法设置参数。

```
Query query=session.createQuery("from Student s where s.age>:minAge and
s.name like:likeName");
```

```
query.setInteger("minAge",25);    //设置:号后的 minAge 变量值
query.setString String("likeName","%clus%");//设置:号后的 likeNam 变量值
```

使用命名参数时, 在建立 Query 时先使用 “:”, 后面带着参数名, 然后就可以在 setxxx()方法中直接指定参数名来设定参数值。命名参数的好处如下:

- 命名参数不依赖于它们在查询字符串中出现的顺序;
- 在同一个查询中可以使用多次;
- 可读性好。

2) setParameter()方法

setParameter()(String paramName, 实例, 实例类型)方法可以绑定任意类型的参数。在实际应用中, Hibernate 可以根据类实例推断出绝大部分对应的映射类型, 因此 setParameter()中的第 3 个参数也可以不要。

【例 5-38】setParameter()方法例子。

```
String hql= "from User u where u.username = ?and u.password = ? ";
Query query=session.createQuery(hql);
query.setParameter(0,username);
query.setParameter(1,password);
```

3) list()方法

Query 的 list()方法用于取得一个 List 类的示例, 此示例中包括的可能是一个 Object 集合, 也可能是 Object 数组集合。

【例 5-39】使用 list()取得一组符合条件的实例对象。

```
Query query=session.createQuery("from Student s where s.age>20");
List list=query.list();
for(int i = 0;i<list.size(); i++){
    Student stu = (Student)list.get(i);
    System.out.println(stu.getName());
}
```

5.3.5 Hibernate的数据关联

在实际项目的数据库设计过程中，往往需要操作多个数据表，而且多个表之间往往存在复杂的关系。下面简单介绍如何在 **Hibernate** 中描述多个表的映射关系，并演示如何操作关系复杂的持久对象。

对象（实体）之间的关联关系分为多对一、一对多、一对一和多对多 4 种情况，本节只介绍前两种关联关系。

1. 多对一

实体与实体间的关系为多对一的关系，在现实中很常见。例如，在学校宿舍管理中，学生作为使用者 **User** 与房间 **Room** 的关系就是多对一的关系，多个使用者可以居住在一个房间，如图 5-34 所示。

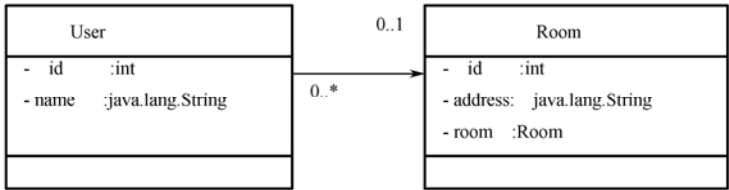


图 5-34 实体多对一关联

【例 5-40】多对一的关系。

可以借 `room_id` 让使用者与房间产生关联，建立如下的 `user` 和 `room` 表格：

```
CREATE TABLE user(
    id INT(11) NOT NULL auto_increment PRIMARY KEY,
    name VARCHAR(100) NOT NULL default "",
    room_id INT(11)
);
```

```
CREATE TABLE room(
    id INT(11) NOT NULL auto_increment PRIMARY KEY,
    address VARCHAR(100) NOT NULL default ""
);
```

用程序来表示的话，`User` 类如下：

```
public class User{
    private Integer id;
    private String name;
    private Room room;
    .....
}
```


User 类中有一个 room 属性，将参考 Room 实例，多个 User 实例可共同参考一个 Room 实例。Room 类代码如下：

```
public class Room{
    private Integer id;
    private String address;
    .....
}
```

映射文件 Room.hbm.xml 代码如下：

```
<class name = "onlyfun.caterpillar.Room" table = "room">
<id name = "id" column = "id">
    <generator class = "native"/>
</id>
<property name = "address" column = "address" type =
    "java.lang.String"/>
</class>
```

很简单的一个映射文件，而在 user.hbm.xml 中，使用<many-to-one>来标志映射多对一关系：

```
<class name = "onlyfun.caterpillar.User" table = "user">
    <id name = "id" column = "id" type = "java.lang.Integer">
        <generator class = "native"/>
    </id>
    <property name = "name" column = "name" type = "java.lang.String"/>
    <many-to-one name = "room"
        column= "room_id"
        class = "onlyfun.caterpillar.Room"
        cascade = "all"
        outer-join = "true"/>
</class>
```

在<many-to-one>的设定中，cascade 表示主控方（User）进行 save-update、delete 等相关操作时，被控方(Room)是否也要进行相关操作。即存储或更新 User 实例时，当中的 Room 实例是否也一起对数据库进行存储或更新操作，如果设定为 all，表示主控方进行任何操作，被控方也进行对应操作。

下面是一个存储的例子：

```
Room room1 = new Room();
Room room2 = new Room();
User user1 = new User();
User user2 = new User();
User user3 = new User();
room1.setAddress("NTU-M8-419");
room2.setAddress("NTU-G3-302");
user1.setName("bush");
user1.setRoom("room1");
user2.setName("Tom");
```

```
user2.setRoom("room2");
user3.setName("Grace");
user3.setRoom("room2");
Session session=sessionFactory.openSession();
Transaction tx = session.beginTransaction();
session.save(user1);           //主控方操作，被控方也对应操作
session.save(user2);
session.save(user3);
tx.commit();
session.close();
```

最后的结果是，room 表格插入了两条记录：

```
room
id    address
1     NTU-M8-419
1     NTU-G3-302

user
id    name    room_id
1     bush    1
2     Tom     2
3     Grace   2
```

在 Hibernate 映射文件中，cascade 属性的设置映射程序的编写，cascade 属性预设值是 none。以多对一的范例来看，如果设定 cascade 不为 true，则必须分别对 User 实例和 Room 实例进行存储，代码如下：

```
session.save(room1);           //存储 Room 实例
session.save(room2);
session.save(user1);           //存储 User 实例
session.save(user2);
session.save(user3);
```

2. 一对多

多对一的关系反过来就是一对多的关系，一对多关系在系统实现中也很常见。典型的例子就是父亲与孩子的关系、房间与使用者的关系。在下面的示例中，每个 Room 都关联到多个 User，即一个房间可以多人居住。

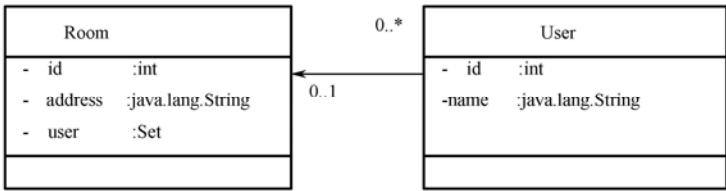


图 5-35 实体一对多关联

【例 5-41】一对多的关系。

User.java 的代码如下：

```
public class User{
    private Integer id;
```

```

        private String name;
        ...
    }

```

而在 Room 类别中，使用 Set 来记录多个 User，代码如下：

```

public class Room{
    private Integer id;
    private String address;
    private Set users;
    ...
}

```

这种方式即所谓单向一对多关系，也就是 Room 实例知道 User 实例的存在，而 User 实例则没有意识到 Room 实例的存在（在多对一中为单向多对一关系，即 User 知道 Room 的存在，但 Room 不知道 User 的存在）。

在映射文件上，user.hbm.xml 代码如下：

```

<class name = "onlyfun.caterpillar.User" table = "user">
    <id name = "id" column = "id" type = "java.lang.Integer">
        <generator class = "native"/>
    </id>
    <property name = "name" column= "name" type = "java.lang.String"/>
</class>

```

在单向关系中，被参考的对象的映射文件就如单一实例一样配置，接下来看看 room.hbm.xml，使用<one-to-many>标志配置一对多。

Room.hbm.xml 的代码如下：

```

<class name = "onlyfun.caterpillar.Room" table = "room">
    <id name = "id" column = "id" >
        <generator class = "native"/>
    </id>
    <property name = "address"
        column= "address"
        type = "java.lang.String"/>
    <set name = "users" table = "user" cascade = "all">
        <key column = "room_id"
            <one-to-many class= "User"/>
        </set>
</class>

```

可以存储实例：

```

User user1 = new User();
User user2 = new User();
User user3 = new User();
Room room1 = new Room();
Room room2 = new Room();
user1.setName("bush");
user2.setName("tom");

```

```
user3.setName("grace");
room1.setUsers(new HashSet());
room1.setAddress("NTU-M8-419");
room1.addUser(user1);
room1.addUser(user2);
room2.setUsers(new HashSet());
room2.setAddress("NTU-G3-302");
room2.addUser(user3);
Session session=sessionFactory.openSession();
Transaction tx = session.beginTransaction();
session.save(room1);      //cascade 操作
session.save(room2);
tx.commit();
session.close();
```

在数据库中将存储以下的表格：

id	name	room_id
1	bush	1
2	tom	1
3	grace	2

id	address
1	NTU-M8-419
2	NTU-G3-302

3. 双向关联

在多对一、一对多的关系中都是单向关联的，也就是其中一方关联到另一方，而另一方不知道自己被关联。如果让双方都意识到另一方的存在，就形成了双向关联，把多对一、一对多的例子改写如下，重新设计 User 类，如图 5-36 所示。

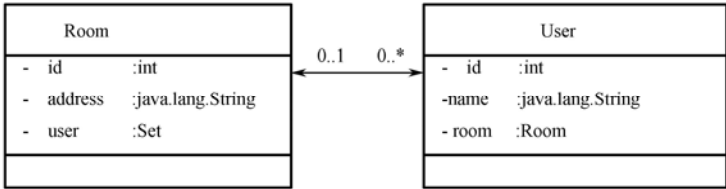


图 5-36 双向关联

【例 5-42】双向关联。

```
public class Use{
    private Integer id;
    private String name;
    private Room room;
    ...
}
```

Room 类如下：

```

public class Room{
    private Integer id;
    private String address;
    private Set users;
    ...
}

```

这样，User 实例可参考 Room 实例而维持多对一关系，而 Room 实例记得 User 实例而维持一对多关系。

映射文件 User.hbm.xml 的代码如下：

```

<class name = "onlyfun.caterpillar.User" table = "user">
    <id name = "id" column = "id" type = "java.lang.Integer">
        <generator class = "native"/>
    </id>
    <property name = "name" column= "name" type = "java.lang.String"/>
    <many-to-one name = "room"
        column = "room_id"
        class = "onlyfun.caterpillar.Room"
        cascade = "save-update"
        outer-join = "true"/>
</class>

```

Room.hbm.xml 的代码如下：

```

<class name = "onlyfun.caterpillar.Room" table = "room">
    <id name = "id"column = "id" >
        <generator class = "native"/>
    </id>
    <property name = "address"column = "address" type = "java.lang.String"/>
    <set name = "users" table = "user" cascade = "save-update">
        <key column = "room_id"/>
        <one-to-many class = "onlyfun.caterpillar.User"/>
    </ set >
</class>

```

映射文件双方都设定了 cascade 为 save-update，所以可以用多对一方式来维持关联：

```

User user1 = new User();
User user2 = new User();
Room room1 = new Room();
user1.setName("bush");
user2.setName("caterpillar");
room1.setAddress("NTU-M8-419");
user1.setRoom(room1);
user2.setRoom(room1);
Session session=sessionFactory.openSession();
Transaction tx = session.beginTransaction();
session.save(user1);
session.save(user2);

```

```

tx.commit();
session.close();
或者反过来由一对多的方式来维持关联:
User user1 = new User();
User user2 = new User();
Room room1 = new Room();
user1.setName("bush");
user2.setName("caterpillar");
room1.setUsers(new HashSet());
room1.setAddress("NTU-M8-419");
room1.addUser(user1);
room1.addUser(user2);
Session session=sessionFactory.openSession();
Transaction tx = session.beginTransaction();
session.save(room1);
tx.commit();
session.close();

```

这里有个效率问题需要讨论，上面程序 **Hibernate** 将使用以下的 SQL 进行存储：

```

Hibernate:insert into room(address)values(?)
Hibernate:insert into user(name,room_id)values(?,?)
Hibernate:insert into user(name,room_id)values(?,?)
Hibernate:update user set room_id = ? where id = ?
Hibernate:update user set room_id = ? where id = ?

```

上面的写法标志关联由 **Room** 单方面维护，而主控方也是 **Room**。**User** 不知道 **Room** 的 **room_id** 是多少，所以必须分别存储 **Room** 与 **User** 之后，再更新 **user** 的 **room_id**。

在一对多、多对一形成双向关联的情况下，可以将关联维持的控制交给多的一方。这样可以提高效率。就像在公司中，老板要记住所有员工的名字慢，每个员工记住老板的名字快。所以在一对多、多对一形成双向关联的情况下，可以在“一”的一方设定控制反转，也就是当存储“一”的一方时，将关联维持的控制权交给“多”的一方。就上面的例子来说，可以设定 **room.hbm.xml**，代码如下：

```

<class name = "onlyfun.caterpillar.Room" table = "room">
    <id name = "id" column = "id" >
        <generator class = "native"/>
    </id>
    <property name="address" column = "address" type = "java.lang.String"/>
    <set name = "users" table = "user" cascade = "save-update" inverse= "true">
        <key column = "room_id"/>
        <one-to-many class = "User"/>
    </set>
</class>

```

由于关联的控制交给“多”的一方，所以直接存储“一”前，“多”的一方必须意识到“一”的存在，所以程序修改如下：

```

User user1 = new User();
User user2 = new User();
Room room1 = new Room();
user1.setName("bush");
user2.setName("tom");
room1.setUsers(new HashSet());
room1.setAddress("NTU-M8-419");
room1.addUser(user1);
room1.addUser(user2);
//多方必须意识到单方的存在
user1.setRoom(room1);
user2.setRoom(room1);
Session session=sessionFactory.openSession();
Transaction tx = session.beginTransaction();
session.save(room1);
tx.commit();
session.close();

```

上面的程序 Hibernate 将使用以下的 SQL:

```

Hibernate:insert into room(address)values(?)
Hibernate:insert into user(name,room_id)values(?,?)
Hibernate:insert into user(name,room_id)values(?,?)

```

5.3.6 Hibernate实体对象生命周期、缓存管理、事务

1. 实体对象的生命周期

实体对象的生命周期是 Hibernate 应用的一个关键概念。对生命周期的理解和把握，不仅对 Hibernate 的正确应用颇有裨益，而且对 Hibernate 实现原理的探索也很有意义。

这里的实体对象，特指 Hibernate O/R 映射关系中的域对象（即 O/R 中的“O”）。

实体对象生命周期有 3 种状态。

1) Transient（瞬时态）

瞬时态，即实体对象在内存中的存在与数据库中的记录无关，例如：

```

User user=new User();
user.setUsername("Tom");

```

这里的 user 对象与数据库中的记录没有任何关联。

2) Persistent（持久态）

持久态是指对象处于由 Hibernate 框架所管理的状态。这种状态下实体对象的引用被纳入 Hibernate 实体容器中加以管理。处于持久态的对象，其变更将由 Hibernate 固化到数据库中。

【例 5-43】持久态。

```

User user=new User();
User anotherUser=new User();
user.setUsername("Tom");           //此时 user 处于瞬时态
anotherUser.setUsername("Grace");   //此时 anotherUser 处于瞬时态

```

```

Transaction tx= session.beginTransaction();
//通过 save()方法, user 对象转换为持久态, 由 Hibernate 纳入实体管理容器
//而 anotherUser 仍然处于瞬时态
session.save(user);
//事务提交之后, 库表中插入一条用户“Tom”的记录
//对于 anotherUser 则无任何操作
tx.commit();

Transaction tx2=session.beginTransaction();
user.setUserName("Tom1");
anotherUser.setUsername("Grace1");
//虽然这个事务中没有显示调用 session.save()方法保存 user 对象
//但是由于 user 为持久态, 将自动被固化到数据库
//因此数据库的用户记录已被更改为“Tom1”
//此时 anotherUser 仍然是一个普通的 Java 对象, 未对数据库产生任何影响
tx2.commit();

```

处于瞬时态的对象, 可以通过 Session 的 save()方法转换成持久态。同样, 如果一个实体对象由 Hibernate 加载, 那么它也处于持久态。

//由 Hibernate 返回的持久对象

```
User user = (User)Session.load(User.class,new Integer(1));
```

持久对象对应数据库中的一条记录, 可以看做数据库记录的对象化操作接口, 其状态的变更将对数据库中的记录产生影响。

3) Detached (脱管态)

处于持久状态的对象, 其对应的 Session 实例关闭后, 此对象就处于脱管态。

Session 实例可以看做持久对象的宿主, 一旦此宿主失效, 其从属的持久对象进入脱管态。

```

//user 处于瞬时态
User user=new User();
User.setUsername("Tom");
Transaction tx = session.beginTransaction();
//user 对象由 Hibernate 纳入管理容器, 处于持久态
session.save(user);
tx.commit();
//user 对象状态为脱管态, 因为与其关联的 session 已经关闭
session.close();

```

上面的例子中, user 对象从瞬时态转变为持久态, 又从持久态转变为脱管态。那么, 这里的脱管态和瞬时态有什么区别呢?

区别在于脱管对象可以再次与某个 Session 实例相关联而成为持久对象。更为重要的是, 当瞬时对象执行 Session.save 方法时, user 对象的内容已经发生了变化。Hibernate 对 user 对象持久化, 并为其赋予了主键值。这个 user 对象可以与库表中具备相同 id 值的记录相关联。瞬时状态的 user 对象与库表中的数据缺乏对应关系。而脱管态的 user 对象, 却在库表中存在相对应的记录, 只不过由于脱管对象脱离 Session 这个数据操作平台, 其状态的改变无法更新到库表中的对应记录。

有时候为了方便, 将处于瞬时态和脱管态的对象统称为值对象 (Value Object, VO), 将

处于持久态的对象称为持久对象（Persistent Object，PO）。这是由“实体对象是否被纳入 Hibernate 实体管理容器”加以区别的，非管理的实体对象统称 VO，被管理的对象称为 PO。

2) 缓存管理

缓存是提高系统性能的重要手段。在大并发量的情况下，如果每次程序都需要从数据库直接做查询操作，它们带来的性能开销是显而易见的。频繁的数据库磁盘读写操作会大大降低系统的性能。此时，如果能让数据库在本地内存中保留一个镜像，下次访问的时候只需要从内存中直接获取即可，显然这样做可以带来性能提升。引入缓存机制的难点是如何保证内存中数据的有效性，否则，脏数据的出现将会给系统带来难以预知的严重后果。对于应用程序，缓存通过内存或磁盘保存了数据库的当前有关数据状态，它是一个存储在本地的数据备份。缓存位于数据库和应用程序之间，从数据库更新数据，并给程序提供数据。

Hibernate 实现了良好的缓存机制，借助 Hibernate 内部的缓存提高数据读取性能。Hibernate 中的缓存可分为两层：一级缓存和二级缓存。

1) 一级缓存

Session 实现了一级缓存，它是事务级数据缓存。一旦事务结束，这个缓存也就失效。一个 Session 的生命周期对应一个数据库事务或程序事务。Session-cache 保证一个 Session 中两次请求同一个对象时，取得的对象是同一个 Java 实例。

2) 二级缓存

二级缓存是 SessionFactory 范围的缓存，所有的 Session 共享一个二级缓存。Session 在进行数据查询操作时，会首先在自身内部的一级缓存中进行查找，如果一级缓存未能命中，则将在二级缓存中查询，如果二级缓存命中，则将此数据作为结果返回。

在引入二级缓存时，需要考虑是否能使用缓存？哪些数据应用二级缓存？显然，数据库中所有的数据都是实施缓存最简单的方法。但是，这样的方式有时反而会对性能造成影响。例如，一个电话务系统，客户可以通过这套系统查询自己的通话记录。对于每个客户，库表中可能都有成千上万条数据，而不同客户之间，基本不可能共享数据（客户只能查询自身的通话记录）。如果对此表施以缓存管理，内存会迅速被几乎不可能重复的数据充斥，系统性能急剧下降。

因此，在考虑缓存机制应用策略的时候，应该对当前系统的数据逻辑进行考查，以确定最佳的解决方案。

在确定了缓存策略后，要挑选一个高效的缓存，它将作为插件被 Hibernate 调用。Hibernate 允许使用下述缓存插件。

EhCache: 可以在 JVM 中作为一个简单进程范围内的缓存，它可以把缓存的数据放入内存或磁盘，并支持 Hibernate 中选用的查询缓存。

OpenSymphony OSCache: 和 EhCache 相似，并且提供了丰富的缓存过期策略。

SwarmCache: 可作为集群范围的缓存，但不支持查询缓存。

JBossCache: 可作为集群范围的缓存，但不支持查询缓存。

OSCache: 可作为集群范围的缓存，能用于 Java 应用程序的普通缓存解决方案。

3. 事务

事务是一个非常重要的概念。本节将讲述 JDBC 事务、JTA 事务的基本概念，以及并发数据库访问过程中要注意的问题。

事务（Transaction）是工作中的基本逻辑单位，可以用于确保数据库能够被正确修改，

避免数据只修改了一部分而导致数据不完整，或者在修改时受到用户干扰。作为一名软件设计师，必须了解事务并合理利用，以确保数据库保存正确完整的数据。

1) 基于 JDBC 的事务管理

Hibernate 是 JDBC 的轻量级封装，本身并不具备事务管理能力。在事务管理层，Hibernate 将其委托给底层的 JDBC 或 JTA，以实现事务管理和调度功能。

在 JDBC 的数据库操作中，一项事务是由一条或者多条表达式组成的不可分割的工作单元。通过提交 `commit()` 或者回滚 `rollback()` 来结束事务的操作。

在 JDBC 中，事务默认是自动提交。也就是说，一条对数据库的更新表达式代表一项事务操作。操作成功后，系统将自动调用 `commit()` 进行提交。否则，将调用 `rollback()` 进行回滚。

在 JDBC 中，可以通过调用 `setAutoCommit (false)` 进行禁止自动提交。之后就可以把多个数据库操作的表达式作为一个事务，在操作完成后调用 `commit()` 来进行整体提交。

将事务管理委托给 JDBC 进行处理是最简单的实现方式，Hibernate 对于 JDBC 事务的封装也比较简单。

看下面这段代码：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
session.save(room);
tx.commit();
```

从 JDBC 层面而言，上面的代码实际上对应着：

```
Connection cn = getConnection();
cn.setAutoCommit(false);
//JDBC 调用相关的 SQL 语句
cn.commit();
```

在 `sessionFactory.open()` 语句中，Hibernate 会初始化数据库连接。与此同时，将其 `AutoCommit` 设为关闭状态 (`false`)。即一开始从 `SessionFactory` 获得的 `session`，其自动提交属性已经被关闭。下面的代码不会对数据库产生任何效果：

```
session session=sessionioin.Factory.openSession();
session.save(room);
session.close();
```

这实际上相当于 JDBC `Connection` 的 `AutoCommit` 属性被设为 `false`，执行了若干 JDBC 操作之后，并没有调用 `commit` 操作。

如果代码真正作用到数据库，必须调用 `Transaction` 指令：

```
Session session= sessionFactory.openSession();
Transaction tx = sessio.beginTransaction();
session.save(room);
tx.commit();
session.close();
```

2) 基于 JTA 的事务管理概念

JTA (Java Transaction API) 是由 Java EE Transaction Manager 负责管理的事务，其最大的特点是：调用 `UserTransaction` 接口的 `begin`、`commit` 和 `rollback` 方法来完成事务范围的界定，以及事务的提交和回滚。JTA Transaction 可以实现同一事务对应不同的数据库。

JTA 主要用于分布式的多个数据源的两阶段提交的事务，而 JDBC 的 `Connection` 提交的

是单个数据源的事务，后者因为只涉及一个数据源，所以其事务可由数据库自己单独实现。而 JTA 事务因为其分布式和多数据源的特性，不可能由任何一个数据源实现事务。因此，JTA 中的事务是由“事务管理器”实现的。它会在多个数据源之间统筹事务，具体使用的技术就是所谓的“两阶段提交”。

JTA 提供了跨 Session 的事务管理能力。这一点是与 JDBC Transaction 最大的差异。JDBC 事务由 Connection 管理，即事务管理实际上是在 JDBC Connection 中实现的。事务周期限于 Connection 的生命周期。同样，对于基于 JDBC Transaction 的 Hibernate 事务管理机制而言，事务管理在 Session 所依托的 JDBC Connection 中实现，事务周期限于 Session 的生命周期。

JTA 事务管理则由 JTA 容器实现。JTA 容器对当前加入事务的众多 Connection 进行调度，实现事务性要求。JTA 的事务周期可横跨多个 JDBC Connection 生命周期。同样，对于基于 JTA 事务的 Hibernate 而言，JTA 事务横跨多个 Session。

3) 锁

业务逻辑的实现过程中，往往需要保证数据访问的排他性。例如，在金融系统的日终结算处理中，希望对某个结算时间点的数据进行处理，而不希望在结算过程中（可能是几秒，也可能是几个小时），数据再发生变化。此时，需要通过一些机制来保证这些数据在某个操作过程中不会被外界修改，这样的机制，就是所谓的“锁”，即给选定的目标数据上锁，使其无法被其他程序修改。

Hibernate 支持两种锁机制：悲观锁（Pessimistic Locking）和乐观锁（Optimistic Locking）。

悲观锁是指对数据被外界修改持保守态度。假定任何时刻存取数据时，都可能有一个客户也正在存取同一数据。为了保持数据被操作的一致性，于是对数据采取了数据库层次的锁定状态。依靠数据库提供的锁机制来实现。

乐观锁则乐观地认为数据很少发生同时存取的问题，因而不做数据库层次上的锁定。为了维护正确的数据，乐观锁采用应用程序上的逻辑实现版本控制的方法。

Hibernate 中通过版本号检索来实现后更新为主，这也是 Hibernate 推荐的方式。在数据库中假如有一个 Version 栏记录，在读取数据时连带版本号一同读取，并在更新数据时递增版本号，然后比较版本与数据库中的版本号，如果大于数据库中的版本号，则给予更新，否则就报错误。

例如，有两个客户端，A 客户先读取了账户余额 200 元，之后 B 客户也读取了账户余额 200 元的数据。在并发情况下，A 客户提取了 100 元，对数据库做了变更，此时数据中的数据余额为 100 元，B 客户也要提取 80 元，根据所取得的资料，200 减去 80 变为 120，再对数据库进行变更，最后的余额就会不正确。

利用乐观锁，A 客户读取账户余额 200 元，并连带读取版本号为 5 的话，B 客户此时也读取账号余额为 200 元，版本号也为 5。A 客户在领款后账号余额为 100，此时将版本号变为 6，而数据库中版本号为 5，所以准予更新。更新数据库后，数据库的余额为 100，版本号为 6。B 客户领款后要变更数据库，其版本号为 5，但是数据库的版本号为 6，此时不予更新。如果 B 客户试图更新数据，将会引发异常。可以捕捉这个异常，在处理数据中重新读取数据库中的数据。

5.3.7 在Web环境下使用Hibernate

本节将讲述在 Web 环境下开发简单的使用 Hibernate 的 Web 应用程序，如简单的 JSP 程序。由于 MyEclipse 5.1 支持 Eclipse 3.2、WTP 及 Hibernate，因此，采用的平台、插件及相关网址如下：

```
J2SE 5.0 JDK:http://java.sun.com/j2se
Eclipse 3.2:http://www.eclipse.org/
Tomcat 5.0.28:http://jakarta.apache.org/tomcat/
MySQL 5.0:http://www.mysql.com/
MySQL Connector/J driver 3.1:http://www.mysql.com/
MyEclipse5.1:http://www.myeclipseide.com
Hibernate Syschronizer:http://www.binamics.com/hibernatesync
```

本节中案例的开发及运行思路：在 Eclipse 下采用 MyEclipse 插件及 Hibernate Synchronizer 插件实现 Web 应用程序的布置、ORM 设计以及 JSP 文件的开发，然后单独在 Tomcat 中部署 Web 应用程序，再在 IE 浏览器或 MyEclipse 的浏览器中访问 JSP 文件。

MyEclipse 5.1 插件的安装和其他插件及平台的安装详见其官方网站，这里不再叙述。

下面简述开发过程：

(1) 在 MySQL 数据库中创建数据库 Hdata1 及表 person。

```
CREATE TABLE 'person' (
    'ID' int (11) NOT NULL auto_increment,
    'Name' varchar (20) collate gb2312_bin NOT NULL default ' ',
    'Sex' char(1) collate gb2312_bin default NULL,
    'Address' varchar(200) collate gb2312_bin default NULL,
    PRIMARY KEY ('ID')
)ENGINE = MyISAM;
```

(2) 创建工程，命名为 Ht5，如图 5-37 所示。

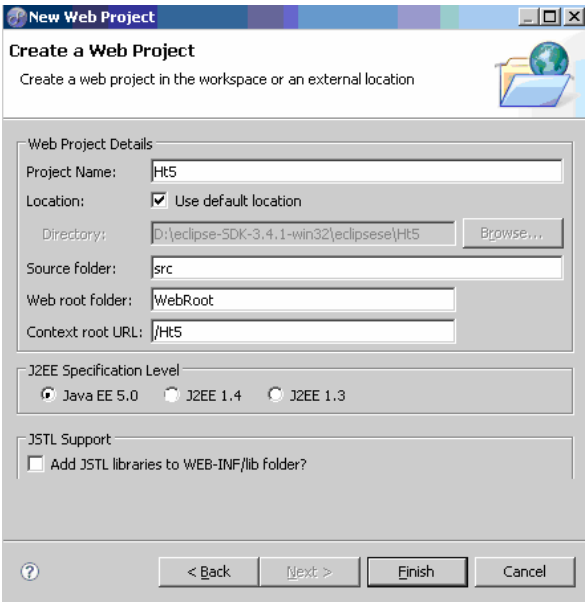


图 5-37 创建工程

(3) 在该工程中套入 MySQL 数据库的连接驱动程序，如 mysql-connector-java-3.1.12-bin.jar。

(4) 在 src 目录中创建包，如 liufy.test，用来存放 POJO。

(5) 用鼠标右键单击工程名，如 Ht5，在弹出的对话框中，利用 MyEclipse 的 Add Hibernate Capabilities 在工程中加入 Hibernate，包括 Hibernate 库文件包、配置文件 hibernate.cfg.xml，如图 5-38~图 5-40 所示。

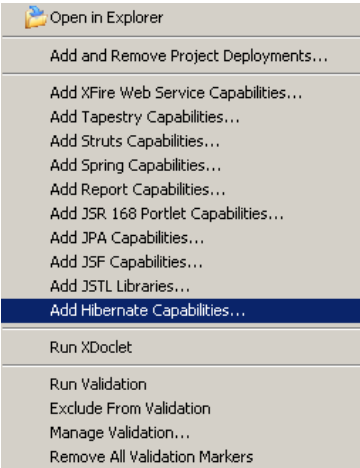


图 5-38 添加 Hibernate 支持

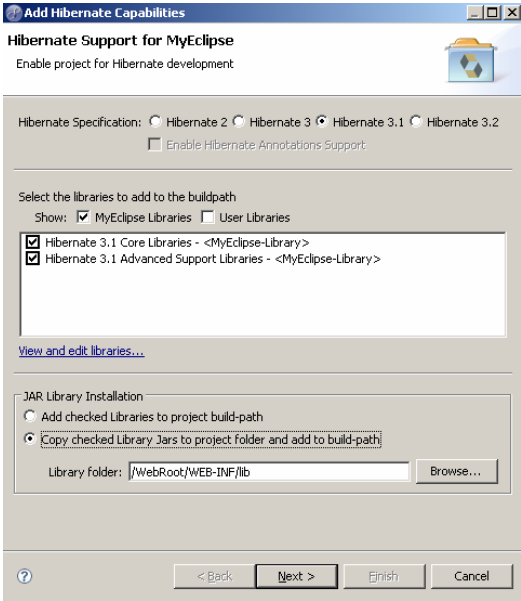


图 5-39 添加 Hibernate 的类库

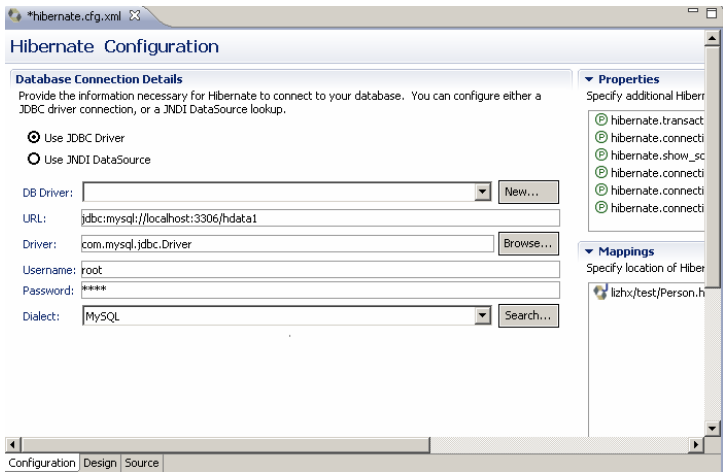


图 5-40 配置 hibernate.cfg.xml

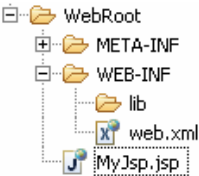


图 5-41 创建 jsp 文件

(6) 在包 liufy.test 下利用 Hibernate Syschronizer 创建 Person 类以及相关的数据库映射配置文件，并修改为采用 DAOs 模式。

(7) 修改配置文件 hibernate.cfg.xml，并加入 POJO 的映射文件。

(8) 在本工程中 Web 发布目录 WebRoot 下编写 JSP 文件，如 MyJsp.jsp，如图 5-41 所示。

(9) 在操作系统中启动 Tomcat，在 IE 浏览器中利用 Tomcat 的管

理工具发布 WebRoot 目录，也可以采用手写 XML 配置文件的方式来发布，如图 5-42 所示。

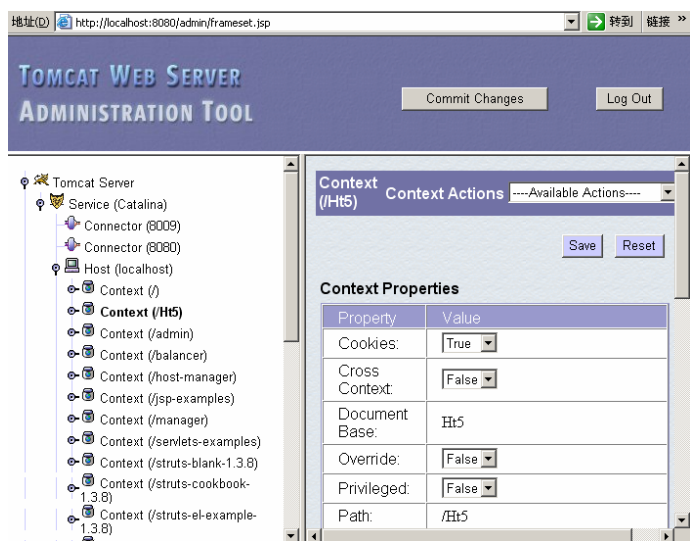


图 5-42 发布网站

(10) 将 MySQL 数据库的连接驱动程序复制到 Tomcat 的公共库中或 WebRoot 目录的 WEB-INF\lib 下。

(11) 重新启动 Tomcat，在 IE 浏览器或 MyEclipse 的浏览器中访问相应的 JSP 文件，如图 5-43 所示。



图 5-43 访问 JSP 文件

【例 5-44】 MyJsp.jsp 文件的内容。

```
<% @ page import = "java.util.*,liufy.test.Person, liufy.test.dao.* "
    errorPage= "error.jsp"%>
<% @ page import = "org.hibernate.*,org.hibernate.criterion.* "%>
<%
try {
    _RootDAO.initialize();
    PersonDAO persondao = new PersonDAO();
    Criteria criteria =
        (persondao.getSession() ).createCriteria(Person.class);
    criteria.add(Expression.eq("Sex", "M" ) );
    List p = criteria.list();
    out.println("The Man persons = "+ p.size() );
} catch(Exception e) {
```

```

        out.println(e.getMessage() );
    }
}
%>

```

如果不采用 DAOs 模式，我们将用到由 MyEclipse 提供的 HibernateSessionFactory 类，如图 5-44 所示。

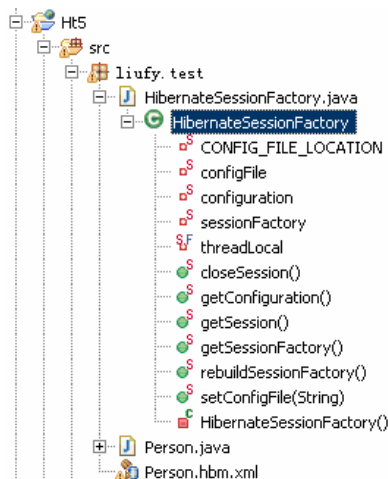


图 5-44 HibernateSessionFactory 类

【例 5-45】 MyJsp.jsp 可以采用的其他形式。

```

<% @ page import = "java.util.*, liufy.test.* " errorPage = "error.jsp"%>
<% @ page import = "org.hibernate.*,org.hibernate.criterion.* "%>
<%
try{
    Criteria criteria = (HibernateSessionFactory.getSession() ).
        CreateCriteria(Person.class);
    criteria.add(Expression.eq("Sex", "M") );
    List p = criteria.list();
    out.println("The Man persons =" +p.size() );
} catch(Exception e) {
    out.println(e.getMessage() );
}
%>
<%HibernateSessionFactory.getSession();
%>

```

【例 5-46】 HibernateSessionFactory.java 的 getSession()方法。

```

public static Session getSession() throws HibernateException {
    Session session = (Session) threadLocal.get();
    if (session == null || ! session.isOpen() ) {
        if(sessionFactory == null) { rebuildSessionFactory(); }
        session = (sessionFactory != null) ?
            sessionFactory.openSession():null;
        threadLocal.set(session);
    }
}

```

```

    }
    return session;
}

```

其中，`threadLocal` 的定义如下：

```

private static final ThreadLocal<Session>threadLocal =
    new ThreadLocal<Session> ();

```

在这段程序中用到了 `threadLocal`。下面简单叙述一下 `threadLocal` 的原理。

`SessionFactory` 负责创建 `Session`，`SessionFactory` 是线程安全的，多个并发线程可以同时访问一个 `SessionFactory`，并从中获取 `Session` 实例。而 `Session` 并非线程安全，也就是说，如果多个线程同时使用一个 `Session` 实例进行数据存取，则将会导致 `Session` 数据存取逻辑混乱。下面是一个典型的 `Servlet`，试图通过一个类变量 `session` 实现 `Session` 的重用，以避免每次操作都要重新创建。

【例 5-47】 一个典型的 `Servlet`。

```

public class TestServlet extends HttpServlet {
    private Session session;
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException,IOException {
        session = getSession();
        doSomething();
        session.flush();
    }
    public void doSomething() {
        //基于 session 的存取操作
    }
}

```

代码看上去正确无误，甚至在单机测试的时候可能也不会发生什么问题，但这样的代码一旦编译部署到实际运行环境中，就会出现莫名其妙的错误。

首先 `Servlet` 运行是多线程的（`JSP` 也是一个 `Servlet`），而应用服务器并不会为每个线程都创建一个 `Servlet` 实例。也就是说，`TestServlet` 在应用服务器中只有一个实例（在 `Tomcat` 中是这样，其他的应用服务器可能有不同的实现），而这个实例会被许多个线程并发调用，`doGet` 方法也将被不同的线程反复调用。可想而知，每次调用 `doGet` 方法，这个唯一的 `TestServlet` 实例的 `session` 变量都会被重置，线程 A 的运行过程中，其他的线程如果也被执行，那么 `session` 的引用将发生改变，之后线程 A 再调用 `session`，可能此时的 `session` 与其之前所用的 `session` 就不再一致，显然，错误也就出现了。

`ThreadLocal` 的出现使得这个问题迎刃而解。`ThreadLocal` 并非是一个线程的本地实现版本，它并不是一个 `Thread`，而是 `ThreadLocal Variable`（线程局部变量）。线程局部变量（`ThreadLocal`）的功能非常简单，就是为每一个使用该变量的线程都提供一个变量值的副本，使每一个线程都可以独立地改变自己的副本，而不会和其他线程的副本冲突。从线程的角度来看，就好像每一个线程都完全拥有该变量。`ThreadLocal` 就从另一个角度来解决多线程的并发访问，`ThreadLocal` 会为每一个线程维护一个和该线程绑定的变量的副本，从而隔离了多个线程的数据。每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行

同步了。`ThreadLocal` 提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的整个变量封装进 `ThreadLocal`，或者把该对象的特定于线程的状态封装进 `ThreadLocal`。

例 5-47 中的程序可以按例 5-48 进行修改。

【例 5-48】使用 `ThreadLocal` 的例子。

```
public class TestServlet extends HttpServlet {
    private ThreadLocal localSession = new ThreadLocal();

    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        localSession.set(getSession() );
        doSomething();
        session.flush();
    }

    public void doSomething() {
        Session session = (Session)localSession.get();
        //基于 session 的存取操作
    }
}
```

其中，`localSession` 是一个 `ThreadLocal` 类型的对象，在 `doGet` 方法中，通过其 `set` 方法将获取的 `session` 实例保存。而在 `doSomething` 方法中，通过 `get` 方法取出 `session` 实例。

这样，`ThreadLocal` 通过以各个线程对象的引用作为区分，从而将不同线程的变量隔离开来。在编写 `Servlet` 程序时，必须注意 `ThreadLocal` 的使用。

5.4 Spring框架

5.4.1 Spring基础及其开发环境

1. Spring概述

Spring Framework（简称 `Spring`）是根据 Rod Johnson 著名的《Expert One-on-One J2EE Design and Development》而开发的 Java EE 应用程序框架。它是一个开源框架，是为了解决企业应用程序开发复杂性而创建的。它是针对 `Bean` 的生命周期进行管理的轻量级容器（`Lightweight Container`）。该框架的主要优势之一就是其分层架构，分层架构允许选择使用某个组件，同时为 Java EE 应用程序开发提供集成的框架。在许多情况下，`Spring` 都能够很好地代换传统的由 Java EE 应用程序服务器所提供的服务。

`Spring` 和 `Struts` 一样都是一种轻量级的 Java EE 应用程序框架，`Struts` 注重的是表现和逻辑耦合的降低，主要是把业务逻辑和表现层分开，但是不涉及业务层与持久层的关联。它是对业务层的层次细化，也就是更深层次地降低了耦合程序。`Spring` 和 `Hibernate` 都对对象（或称实体）的关系进行了管理，但是实际上它们的方式和侧重点都不相同。`Spring` 的关系管理是面向业务的，它对对象关系的管理主要涉及到依赖、动态行为管理和行为封装，由此产生了 `IoC`、`AOP`、`BeanFactory` 等机制。也就是说，相对于 `Hibernate` 主要关注于对象（或者在这个范围内应该称之为实体）的数据关系的管理，`Spring` 主要关注的是对象的行为关系

的管理。两者一个是静态关系管理，一个是动态关系管理。

Spring 的核心是个轻量级的容器，它是实现 IoC（Inversion of Control，控制反转）容器、非侵入性（No Intrusive）的框架，并提供 AOP（Aspect-Oriented Programming，面向方面的编程）概念的实现方式，提供对持久层（Persistence）、事务（Transaction）的支持，提供 MVC 框架的实现，并对一些常用的企业服务 API（Application Interface）提供一致的模型封闭，是一个全方位的应用程序框架（Application Framework）。除此之外，对于现存的各种框架（Struts、JSF、Tapestry、Hibernate 等），Spring 也提供了与它们相整合的方案，如图 5-45 所示。

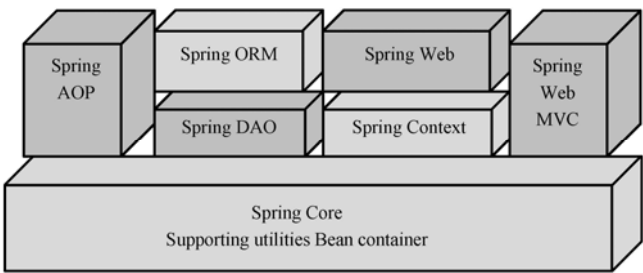


图 5-45 Spring 组成模块

图 5-45 中每个模块的功能如下：

（1）Spring Core: Spring Core（核心容器）提供 Spring 框架的基本功能。核心容器的主要组件是 BeanFactory，它是工厂模式的实现。BeanFactory 使用控制反转（IoC）模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。

（2）Spring Context: Spring Context（Spring 上下文）是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，例如，JNDI、EJB、电子邮件、国际化、校验和调度功能。

（3）Spring AOP: 通过配置管理特性，Spring AOP 模块直接将面向方面的编程功能集成到了 Spring 框架中。Spring AOP 模块为基于 Spring 的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP，不用依赖 EJB 组件，就可以将声明性事务管理集成到应用程序中。

（4）Spring DAO: JDBC DAO 抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量（如打开和关闭连接）。Spring DAO 的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。

（5）Spring ORM: Spring 框架插入了若干个 ORM 框架，从而提供了 ORM 的对象关系工具，其中包括 JDO、Hibernate 和 iBatis SQL Map，所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。

（6）Spring Web: Web 上下文模块建立在应用程序上下文模块之上，为基于 Web 的应用程序提供了上下文。所以，Spring 框架支持与 Jakarta Struts 的集成。Web 模块还简化了处理多部分请求，以及将请求参数绑定到域对象的工作。

（7）Spring Web MVC: MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口，MVC 框架变成为高度可配置的框架。

从图 5-45 上可以看到, Spring Framework 的模块性很强, 用户可以选择使用它的一个或多个功能, 例如, 仅使用它的 BeanFactory/IoContainer, 甚至只使用它对 JDBC 的封装。如果开发团队中比较熟悉 Struts, 则可以继续使用 Struts 作为 Web 框架, 而使用 Spring 提供的 JavaBean 管理和事务支持。使用多少完全是开发人员的选择, 这一点上不像一些其他的技术, 要么不用, 要么都用。

IoC (控制反转) 是用配置文件 (XML) 来描述类与类之间的关系, 由容器来管理, 降低了程序间的耦合度, 程序的修改可以通过简单的配置文件修改来实现。AOP (面向方面的编程) 是一种新兴的编辑技术。它可以解决 OOP 和过程化方法不能够很好地解决的横切 (Crosscut) 问题, 如事务、安全、日志等横切关注。Spring 抽象服务借助于各种 Java EE API 抽象, 把各种不同的 Java EE API 统一起来。这样, 开发者能够迅速掌握各种 Java EE API 的核心内容, 能够一致地使用 Java EE 技术, 减少应用代码量, 精简系统。

Spring IoC+Spring AOP+Spring 抽象服务形成了有机的 Spring。这样一个有机的整体使得构建轻量级的 Java EE 框架成为可能。这 3 个组件在一起工作非常有效。但是一旦剥离任何其中的一个, 其整体性能就会大打折扣。这正是因为 Spring 提供了一套完整的工具, 才使得它倍受开发者推崇, 这也是它与同类其他 Web 框架的区别所在。

2. Spring 的开发环境

1) Spring 的下载

Spring 同时提供了二进制的发布版和相应的源代码。开发者通过 Source Forge 的网站 <http://sourceforge.net/projects/springframework> 就可以获得这些文件, 如图 5-46 所示。

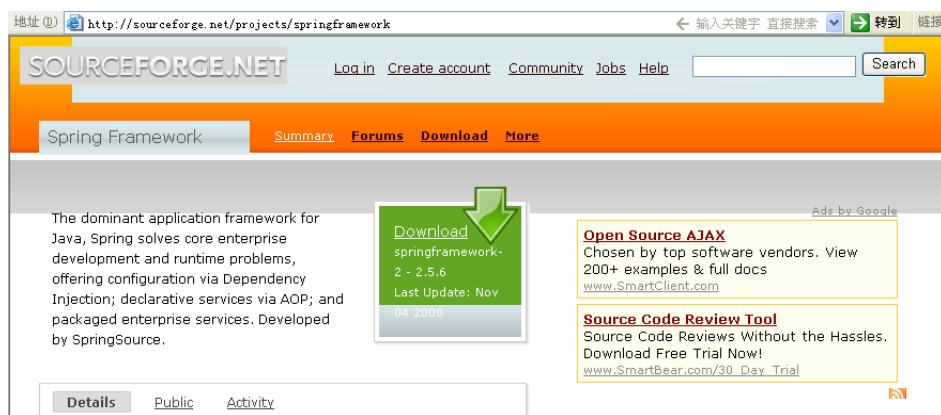


图 5-46 下载 Spring Framework

Spring 二进制发布版提供两种版本。一种是 Spring 二进制发布版本本身, 它不提供其所依赖的第三方库, 如 `spring-framework-1.2.8.zip`。另一种包含了 Spring 二进制发布版所依赖的第三方库, 如 `spring-framework-1.2.8-with-dependencies.zip`。推荐下载后者, 同时也可采用 MyEclipse 插件中自带的 `spring-framework-1.2` 版本。

2) Spring 的包文件

下载 `spring-framework-1.2.8-with-dependencies.zip`。如果读者已经清楚并已配置了 Spring 所依赖的相关开源包, 单独下载 `spring-framework-1.2.8.zip` 即可。下面简单介绍这些包的内容及相关关系。

Spring-framework-1.2.8-with-dependencies.zip 解压后的目录结构，如图 5-47 所示。

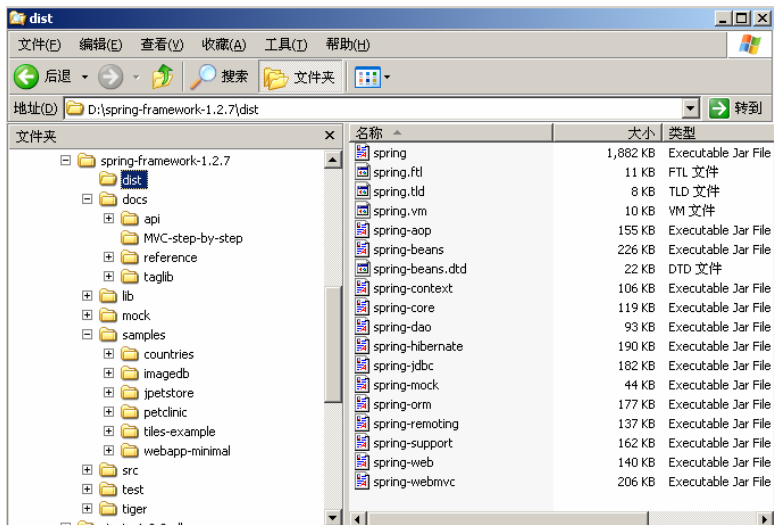


图 5-47 Spring 目录结构

(1) Spring 发布文档目录结构。

① dist 目录下是 Spring 的发布包。

② docs 目录下是相关的文档，包括有 Spring api 的 javadoc、reference 参考指南、Spring 的 taglib 标签使用文件及 Spring MVC 的 MVC-step-by-step 讲解与示例。

③ lib 目录下是 Spring 所依赖的第三方开源包。

④ mock 目录下是 Spring 辅助应用测试的 Mock 源程序。

⑤ samples 目录下是 Spring 的示例源程序及简单的 webapp 示例框架的示例配置。

⑥ src 目录下是 Spring 的源程序。

⑦ test 目录下是 Spring 的单元测试源程序。

⑧ tiger 目录下是 Java 1.5 Tiger 方面的相关及测试源程序。

(2) Spring 包结构说明。spring.jar 包含有完整发布的单个 JAR 包，spring.jar 中包含了除 spring-mock.jar 里包含的内容外其他所有 JAR 包的内容。因为只有在开发环境下才会用到 spring-mock.jar 来进行辅助测试，正式应用系统中是用不到这些类的。

除了 spring.jar 文件，Spring 还包括有其他 13 个独立的 JAR 包（spring-core.jar、spring-bean.jar、spring-aop.jar、spring-context.jar、spring-hibernate.jar、spring-dao.jar、spring-jdbc.jar、spring-orm.jar、spring-remoting.jar、spring-support.jar、spring-web.jar、spring-webmvc.jar、spring-mock.jar），各自包含着对应的 Spring 组件，用户可以根据自己的需要来选择组合自己的 JAR 包，而不必引入整个 spring.jar 的所有类文件。

如何选择这些发布包，决定选用哪些发布包其实相当简单。如果用户正在构建 Web 应用并将全程使用 Spring，那么最好就使用单个全部的 spring.jar 文件；如果用户的应用仅仅用到简单的 IoC/DI 容器，那么只需 spring-core.jar 与 spring-bean.jar 即可；如果用户对发布的大小要求很高，那么就得精挑细选了，只取包含自己所需特性的.jar 文件。采用独立的发布包，用户要以避免包含自己的应用不需要的全部类。当然，用户可以采用其他的一些工具来设法令整个应用包变小，节省空间的重点在于准确地找出自己所需的 Spring 依赖类，然

后合并所需的类与包即可。Eclipse 有个插件叫 ClassPath Helper，可以帮助用户找到所依赖的类。

3) Spring 的辅助工具

在 Spring 应用程序的开发中需要对 Bean 进行配置，有一些 XML 文件需要编写。建议初学者在 Eclipse 中手工进行配置。待熟悉后可以采用一些工具，如 Spring IDE 以及 MyEclipse 插件。

Spring IDE 是 Spring 官方网站所推荐的 Eclipse plug-in，可提供在开发 Spring 时对 Bean 定义文件进行验证，以可视化的方式观看 Bean 与 Bean 之间的依赖关系等功能。

在安装 Spring IDE 之前，必须先安装 Graphical Editing Framework。因为 Spring IDE 使用它作为基础以显示 Bean Graph，也就是以图形化方式显示 Bean 与 Bean 之间的依赖关系。

另外，在 MyEclipse 插件中也整合了 Spring。读者可自己下载和安装该插件，安装时注意 Eclipse 的版本。

5.4.2 Spring的IoC、容器及基本配置

1. IoC (Inversion of Control, 控制反转)

传统模式中类和类之间是直接调用的，所以有很强的耦合度，程序之间的依赖关系比较强，后期维护时牵扯得比较多，而 IoC 用配置文件（XML）来描述类与类之间的关系，由容器来管理，降低了程序间的耦合度，程序的修改可以通过简单的配置文件修改来实现。

在 Spring 中所有的类都会在 Spring 容器中登记，告诉 Spring 用户是谁，用户需要什么，然后 Spring 会在系统运行到适当的时候，把用户要的内容主动发送。所有的类的创建、销毁都由 Spring 来控制。也就是说，控制对象生存周期的不再是引用它的对象，而是 Spring。对于某个具体的对象而言，以前是它控制其他对象，现在是所有对象都被 Spring 控制。这就是控制反转的概念。

IoC 动态地向某个对象提供它所需要的其他对象，这一特点是通过 DI (Dependency Injection, 依赖注入) 来实现的。例如，对象 A 需要操作数据库，以前总是要在 A 中自己编写代码来获得一个 Connection 对象。在 Spring 中，只需要通过配置文件告诉 Spring，A 中需要一个 Connection，至于这个 Connection 怎么构造，何时构造，A 不需要知道。在系统运行时，Spring 会在适当的时候制造一个 Connection，然后像打针一样，注射到 A 当中，这样就完成了对各个对象之间关系的控制。A 需要依赖 Connection 才能正常运行，而这个 Connection 是由 Spring 注入到 A 中的，这就是依赖注入的概念。JDK 1.3 之后一个重要特征是反射 (Reflection)，它允许程序在运行的时候动态地生成对象，执行对象的方法，改变对象的属性，Spring 就是通过反射来实现注入的。

总之，就 Spring 而言，就是通过配置文件，让 Spring 如同一个管家一样来管理所有的 Bean 类。Spring 的依赖注入主要是调用别的 Bean，不是通过实例化对象来调用，而是告诉 Spring，用户需要什么 Bean，然后 Spring 再向用户的 Bean 里面注入用户所需要的 Bean 对象。

下面用代码简单说明。在下面的代码中，Business 的变量 id 可以接收任何 Idependency 的实例，而 Dependency 的实例不是通过 Business 来获得的，而是通过 setter（也可以用构造器）来由外部传给它。例 5-49 中创建了一个 SpringFirst.xml，进行简单的配置。

【例 5-49】 一个 IoC 简例。

```
<bean>
<bean id = "dependency" class = "aopfirst.business.Dependency"/>
<bean id = "business" class = "aopfirst.business.Business">
    <property name = "dependency">
        <ref bean = "dependency"/>
    </property>
</bean>
</bean>
```

这个配置文件里加入了 `Dependency` 类和 `Business` 类，并将 `Dependency` 标记为 `Business` 的一个参数。

有了上面的这个配置文件还不够，还需要一个测试类来加载配置 XML 文件。`Spring` 提供了现成的 API，再加载上面的 XML：实例化 `Dependency` 类、实例化 `Business` 类，并将 `Dependency` 的实例作为参数赋给了 `Business` 实例的 `setDependency()` 方法。下面是该测试程序：

```
public class StartServer {
    public static void main (String [ ] args) {
        ClassPathResource cr = new ClassPathResource("SpringFirst.xml");
        BeanFactory factory = new XmlBeanFactory(cr);
        Business b = (Business)factory.getBean("business");
        b.doSth();
    }
}
```

上面的程序加载了 XML 文件以后，获得 id 为 “business” 的 bean，即 `Business` 类的实例，并调用了其 `doSth()` 方法。由此可见，`Business` 的依赖类 `Dependency` 是通过 XML 来注入的，而且 `Business` 通过接口 `IDependency` 来接收 `Dependency` 实例。因此，当我们又有新的 `IDependency` 的实现时，只需要修改 XML 文件即可，测试程序只需要根据 XML 里的 id 值来获得需要的参数。

2. 用 MyEclipse 支持创建 Spring 工程

下面将使用 `MyEclipse` 插件中的 `Spring` 支持来创建 `Web` 工程。该过程和创建 `Hibernate` 及 `Struts` 很相似，该工程的创建过程中将自动生成 `Web` 应用程序下的一些目录及配置文件。

【例 5-50】 用 `MyEclipse` 支持创建 `Spring` 工程。

- (1) 在 `MyEclipse` 中创建 `Web` 工程，命名为 `springtest2`。
- (2) 在该工程添加 `Spring` 支持。添加 `Spring` 支持是使用 `MyEclipse` 提供的功能来实现的。

① 单击新建 `springtest2` 工程的文件夹，然后单击 “`MyEclipse`” → “`Add Spring Capabilities`” 命令，弹出添加 `Spring` 支持的对话框，如图 5-48 所示。如果只使用 `Spring` 的基本功能，可以不选择 “`Web`” 选项。

② 创建 `Spring` 的配置文件。如图 5-49 所示，“`Folder`” 文本框用来设定 `Spring` 配置文件的放置路径，“`File`” 文本框用来设定配置文件的文件名。本工程中，配置文件的放置路

径为 WebRoot/WEB-INF，配置文件名为 applicationContext.xml。

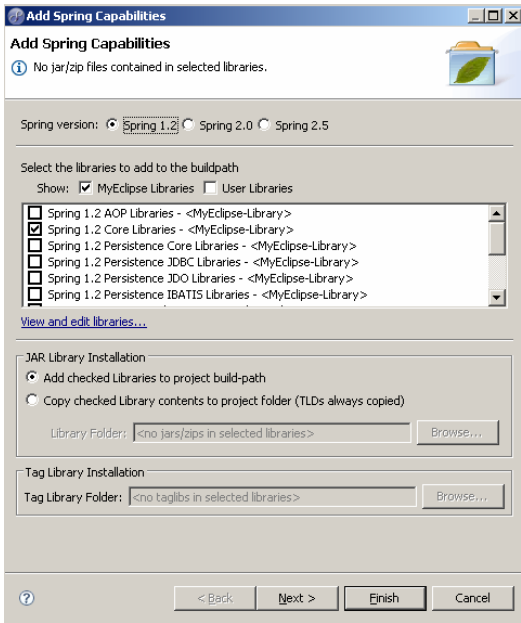


图 5-48 选择 Spring 库

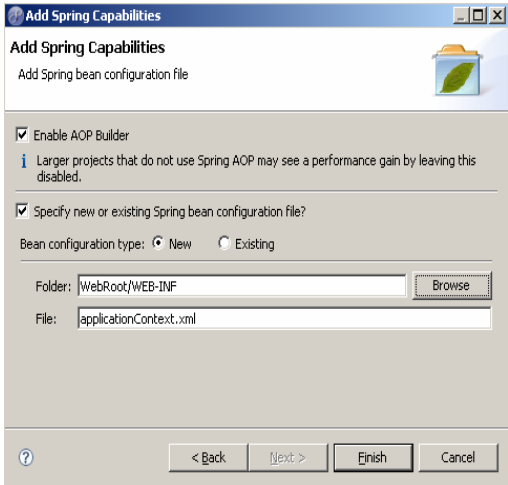


图 5-49 创建配置文件及其路径

（3）编写配置文件和 JavaBean 在 src 目录下创建 liufy.test 包，在该包下编写 ShowMessage 类。其内容如下：

```
package liufy.test;

public class ShowMessage {
    public void show () {
        System.out.println("Hello world");
    }
}
```

然后在配置文件 applicationContext.xml 中编写配置信息：

```
<? Xml version = "1.0" encoding = "UTF-8"?>
<! DOCTYPE bean PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-bean.dtd">
<bean>
    <bean id = "message" class = "liufy.test.ShowMessage">
    </bean>
</bean>
```

（4）编写测试文件 Tests.java。测试文件的内容如下：

```
package liufy.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class Tests {

    public static void main(String args[ ]) {

        ApplicationContext ctx = new FileSystemXmlApplicationContext(
            "WebRoot/WEB-INF/applicationContext.xml");

        //从上下文环境中获取 myBean
```

```

        ShowMessage sm = (ShowMessage)ctx.getBean("message");
        //调用 ShowMessage 的 show 函数输出消息
        sm.show();
    }
}

```

该工程的目录树结构如图 5-50 所示。

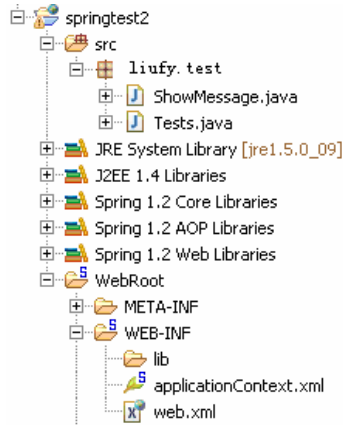


图 5-50 工程 springtest2 目录树结构

(5) 运行程序。把 `Tests.java` 作为应用程序运行，在控制台会输出“Hello World”字符串。需要注意的是，如果在 Web 环境下运行该程序，程序中关于配置文件的位置将改成如下形式：

```
WEB-INF/applicationContext.xml
```

3. Spring 容器

容器是 Spring 框架的核心，Spring 容器使用 IoC 管理所有组成应用系统的组件。Spring 有两种不同的容器：BeanFactory（Bean 工厂，由 `org.springframework.beans.factory.BeanFactory` 接口定义）是最简单的容器，提供了基础的依赖注入支持；ApplicationContext（应用上下文，由 `org.springframework.context.ApplicationContext` 接口定义）建立在 Bean 工厂基础之上，提供了系统构架服务，如从属性文件中读取文本信息，向有关的事件监听器发布事件。

1) BeanFactory

BeanFactory 采用了工厂设计模式。这个类负责创建和分发 Bean，但是不像其他工厂模式的实现，它们只是分发一种类型的对象，而 Bean 工厂是一个通用的工厂，可以创建和分发各种类型的 Bean。

在 Spring 中有几种 BeanFactory 的实现，其中最常使用的是 `org.springframework.bean.factory.xml.XmlBeanFactory`。它根据 XML 文件中的定义装载 Bean。

要创建 `XmlBeanFactory`，需要传递一个 `java.io.InputStream` 对象给构造函数。`InputStream` 对象提供 XML 文件给工厂。例如，下面的代码片段使用一个 `java.io.FileInputStream` 对象把 Bean XML 定义文件给 `XmlBeanFactory`：

```

BeanFactory factory=new XmlBeanFactory(new FileInputStream("beans.xml"));

```

这行简单的代码告诉 Bean 工厂从 XML 文件中读取 Bean 的定义信息，但是现在 Bean

工厂没有实例化 Bean，Bean 被延迟载入到 Bean 工厂中，就是说 Bean 工厂会立即把 Bean 定义信息载入进来，但是 Bean 只有在需要的时候才被实例化。

为了从 BeanFactory 得到一个 Bean，只要简单地调用 `getBean()` 方法，把需要的 Bean 的名字当做参数传递进去即可。

```
MyBean myBean= (MyBean)factory.getBean("myBean");
```

当 `getBean()` 方法被调用的时候，工厂就会实例化 Bean，并使用依赖注入开始设置 Bean 的属性。这样就在 Spring 容器中开始了 Bean 的生命周期。

2) ApplicationContext

BeanFactory 对简单应用来说已经很好了，但是为了获得 Spring 框架的强大功能，需要使用 Spring 的更加高级的容器——ApplicationContext，应用上下文。

表面上，ApplicationContext 和 BeanFactory 差不多。两者都是载入 Bean 定义信息，装配 Bean，根据需要分发 Bean，但是 ApplicationContext 提供了更多功能。

(1) 应用上下文提供了文本信息解析工具，包括对国际化的支持；

(2) 应用上下文提供了载入文本资源的通用方法，如载入图片；

(3) 应用上下文可以向注册为监听器的 Bean 发送事件。

由于它提供了许多附加功能，几乎所有的应用系统都选择 ApplicationContext，而不是 BeanFactory。

在 ApplicationContext 的诸多实现中，有三个常用的实现。

(1) **ClassPathXmlApplicationContext**：从类路径中的 XML 文件载入上下文定义信息，把上下文定义文件当成类路径资源。

(2) **FileSystemXmlApplicationContext**：从文件系统中的 XML 文件载入上下文定义信息。

(3) **XmlWebApplicationContext**：从 Web 系统中的 XML 文件载入上下文定义信息。

【例 5-51】在 ApplicationContext 的诸多实现中的三个常用的实现。

```
ApplicationContext context=new ClassPathApplicationContext("foo.xml");
```

```
ApplicationContext context=new FileSystemXmlApplicationContext  
    ("c:/foo.xml");
```

```
ApplicationContext context=WebApplicationContextUtils.getWebApplicationContext  
    (request.getSession().
```

```
getServletContext());
```

使用 **FileSystemXmlApplicationContext** 和 **ClassPathXmlApplicationContext** 的区别是：**FileSystemXmlApplicationContext** 只能在指定的路径中寻找 `foo.xml` 文件，而 **ClassPathXmlApplicationContext** 可以在整个类路径中寻找 `foo.xml`。

除了 ApplicationContext 提供的附加功能外，ApplicationContext 与 BeanFactory 的另一个重要区别是关于单实例 Bean 如何被加载。Bean 工厂延迟载入所有的 Bean，直到 `getBean()` 方法被调用时，Bean 才被创建。ApplicationContext 则会在上下文启动后预载入所有的单实例 Bean。通过预载入单实例 Bean，确保当需要的时候它们已经准备好了，应用程序不需要等待它们被创建。

4. Spring基本配置

在 Spring 容器内拼凑 Bean 叫做装配。装配 Bean 的时候，是在告诉容器需要哪些 Bean 以及容器如何使用依赖注入，将它们配合起来。

1) 使用 XML 装配

理论上, Bean 装配可以从任何配置资源获得, 但实际上, XML 是最常见的 Spring 应用系统配置源。

【例 5-52】 XML 文件展示了一个简单的 Spring 上下文定义文件。

```
<?xml version= "1.0" encoding= "UTF-8"?>
...
<beans>                                //根元素
    <bean id= "foo"class= "com.spring.Foo"/>    //Bean 实例
    <bean id= "bar"class= "com.spring.Bar"/>    //Bean 实例
</beans>
```

所有 XML 文件定义 Bean, 上下文定义文件的根元素<beans>。<beans>有多个<bean>子元素。每个<bean>元素定义了一个 Bean (任何一个 Java 对象) 如何被装配到 Spring 容器中。

这个简单的 Bean 装配 XML 文件在 Spring 中配置了两个 Bean, 叫做 foo 和 bar。

2) 添加一个 Bean

在 Spring 中对一个 Bean 的最基本配置包括 Bean 的 id 和它的全称类名。向 Spring 容器中添加一个 Bean 只要求向 XML 文件中添加一个<bean>元素。

```
<bean id= "foo"class= "com.spring.Foo"/>
```

(1) 原型与单实例。Spring 中的 Bean 默认情况下是单实例模式。在容器分配 Bean 的时候, 它总是返回同一个实例。但是, 如果想每次向 ApplicationContext 请求一个 Bean 的时候总是得到一个不同的实例, 需要将 Bean 定义为原型模式。

<bean>的 singleton 属性告诉 ApplicationContext 该 Bean 是不是单实例 Bean, 默认为 true, 但是把它设置为 false 的话, 就把该 Bean 定义成了原型 Bean。

```
<bean id= "foo" class= "com.spring.Foo"singleton= "false"/>    //原型 Bean
```

(2) 实例化与销毁。当一个 Bean 实例化的时候, 有时需要做一些初始化的工作才能使用。同样, 当 Bean 不再需要, 可从容器中删除的时候, 需要按顺序做一些清理工作。因此, Spring 可以在创建和拆卸 Bean 的时候调用 Bean 的两个生命周期方法。

在 Bean 的定义中设置自己的 init-method, 这个方法在 Bean 被实例化的时候马上被调用, 同样, 也可以设置自己的 destroy-method, 这个方法在 Bean 从容器中删除之前调用。

【例 5-53】 一个典型的例子——连接池 Bean。

```
public class MyConnectionPool{
    ...

    public void initialize(){//initialize connection pool}
    public void close(){//release connection}
    ...
}
```

Bean 的定义如下:

```
<bean id= "connectionPool" class= "com.spring.MyConnectionPool"
    init-method= "initialize"    //当 Bean 被载入容器的时候调用 initialize 方法
    destroy-method= "close"/>    //当 Bean 从容器中删除的时候调用 close 方法
```

按上述配置, MyConnectionPool 被实例化后, initialize() 方法马上被调用, 给 Bean 初始化连接的机会。在 Bean 从容器中删除之前, close() 方法将释放数据库连接。

3) 通过 Set 方法注入依赖

Set 注入依赖是一种基于标准命名规范的设置 Bean 属性的技术。JavaBean 规范规定使用对应的 set 和 get 方法来设置和获得 Bean 的属性值(Spring 还提供了另外一种依赖注入方式: 构造函数注入)。

【例 5-54】通过 set 方法注入依赖。

```
public class Business{
    IWriter writer;
    ...
    public void setWriter(IWriter writer) {
        this.writer=writer;
    }
    public IWriter getWriter(){
        return writer;
    }
}
```

<bean>元素的子元素<property>指明了使用 set 方法来注入。在<property>元素中, 可以定义装配的属性以及要注入的值, 可以注入任何对象, 从基本类型到集合类, 甚至应用系统中的 Bean。

(1) 简单 Bean 装配。Bean 通常会有一些简单类型属性, 如 int 和 String, 通过使用<property>的子元素<value>可以按照下面的方式设置基本类型属性, 如 int、float 或 java.lang.String。

【例 5-55】Foo 类和配置 Bean。

```
public class Foo{
    String name;
    ...
    public void setName (String name) {
        this.name=name;
    }
    ...
}
```

可以这样配置 Bean:

```
<bean id= "foo"class= "com.spring.Foo">
    <property name "name">
        <value>Foo McFoo</value> //通过 setName("Foo McFoo")方法来设置 name 属性
    </property>
</bean>
```

(2) 应用其他 Bean。通过与 Bean 定义相同的方式, 容器 XML 文件知道应用程序中的其他 Bean。使用<property>元素来设置指向其他 Bean 的属性和<property>的子元素<ref>的实现。

【例 5-56】应用其他 Bean。

```
public interface IWriter(){
    public void save();
}

public class UsbDiskWriter implements(){
```

```

        public void save(){    ...        //存至 USB Disk 中    }
    }
    public class FloppyWriter implements(){
        public void save(){    ...        //存至 Floppy 中    }
    }
    public class Business{
        IWriter writer;
        public setWriter(IWritr writer){
            this.writer=writer;
        }
        public IWriter getWriter(){
            return writer;
        }
        public void save(){writer.save();}
    }

```

对应的 Bean 装配:

```

<bean id= "usbDiskWriter"class= "UsbDiskWriter"/>
<bean id= "floppyWriter"class= "FloppyWriter"/>
<bean id= "business"class= "Business"/>
    <property name= "writer">
        <ref bean= "usbDiskWriter"/>    //装配名字叫 usbDiskWriter 的 Bean, 根据需
            要, 也可以修改成名字为 floppyWriter 的 Bean, 便于实现不同介质的存储
    </property>
</bean>

```

5.4.3 Spring的AOP

面向方面编程（AOP，Aspected Oriented Programming）是面向对象编程（OOP）的一种扩展技术，能够很好地解决横切关注点问题以及相关的设计难题来实现松散耦合。Spring AOP 是 AOP 技术的一种实现。本节将介绍 AOP 的概念，以及 Spring 的 AOP 的实现。

1. 从代理机制初识AOP

从一个简单的例子开始认识 AOP。这个例子中含有日志操作，程序中经常需要记录某些动作或事件，以便随时检查程序运作过程、排除错误的信息。

【例 5-57】当需要在执行某些方法时留下日志信息的一个简单的例子。

```

import java.util.logging.*;
public class HelloSpeaker{
    private Logger logger=Logger.getLogger(this.getClass().getName());
    public void hello(String name){
        logger.log(Level.INFO, "hello method starts...");//方法执行开始时留下日志
        System.out.println("hello, "+name);                //程序的主要功能
        Logger.log(Level.INFO, "hello method ends...");//方法执行完毕时留下日志
    }
}

```

在 `HelloSpeaker` 类中，当执行 `hello()` 方法时，程序员希望开始执行该方法与执行完毕时都留下日志，最简单的做法是如例 5-57 中的程序，在方法执行的前后加上日志动作。然而对于 `HelloSpeaker` 来说，日志的这种动作并不属于 `HelloSpeaker` 逻辑，这使得 `HelloSpeaker` 增加了额外的职责。

如果程序中这种日志动作到处都有需求，以上的写法势必造成程序员必须到处撰写这些日志动作的代码。这将使得维护日志代码的困难加大。如果需要的服务不只是日志动作，有一些非类本身职责的相关动作也混入到类中，如权限检查、事务管理等，会使得类的负担加重，甚至混淆本身的职责。

另一方面，使用以上的写法，如果有一天不再需要日志（或权限检查、交易管理等）的服务，将需要修改所有留下日志动作的程序，无法简单地将这些相关服务从现有的程序中移除。

可以使用代理（Proxy）机制来解决这个问题，有两种代理方式：静态代理（static proxy）和动态代理（dynamic proxy）。

在静态代理的实现中，代理类与被代理必须实现同一个接口，在代理类中可以实现记录等相关服务，并在需要的时候再呼叫被代理类。这样被代理类中就可以仅仅保留业务相关的职责了。

【例 5-58】 一个静态代理的简单例子。

首先定义一个 `Ihello` 接口。`Ihello.java` 的代码如下：

```
public interface Ihello{
    public void hello(String name);
}
```

然后让实现业务逻辑的 `HelloSpeaker` 类实现 `Ihello` 接口，`HelloSpeaker.java` 的代码如下：

```
public class HelloSpeaker implements Ihello{
    public void hello(String name){
        System.out.println("hello, "+name);
    }
}
```

可以看到，在 `HelloSpeaker` 类中没有任何日志的代码插入其中，日志服务的实现将被放到代理类中，代理类同样要实现 `Ihello` 接口。

`HelloProxy.java` 的代码如下：

```
public class HelloProxy implements Ihello{
    private Logger logger=Logger.getLogger(this.getClass().getName());
    private Ihello helloObject;
    public HelloProxy(Ihello helloObject){
        this.helloObject = helloObject;
    }
    public void hello(String name) {
        log("hello method starts...");           //日志服务
        helloObject.hello(name);                 //执行业务逻辑
        log("hello method ends...");             //日志服务
    }
    private void log(String ms){
```

```

        logger.log(Level.INFO,msg);
    }
}

```

在 `HelloProxy` 类的 `hello()`方法中，真正实现业务逻辑前后可以安排记录服务，可以实际撰写一个测试程序来看看如何使用代理类。

```

public class ProxyDemo{
    public static void main(String[ ] args) {
        Ihello proxy=new HelloProxy(new HelloSpeaker() );
        proxy.hello("Justin");
    }
}

```

这是静态代理的基本示例，但是可以看到，代理类的一个接口只能服务于一种类型的类，而且如果要代理的方法很多，势必要为每个方法进行代理，静态代理在程序规模稍大时必定无法胜任。

2. 动态代理

在 `JDK1.3` 之后加入了可协助开发动态代理功能的 `API` 等相关类别，不需要为特定类和方法编写特定的代理类，使用动态代理，可以使得一个处理者（`Handler`）为各个类服务。

要实现动态代理，同样需要定义所要代理的接口。

【例 5-59】 一个动态代理的简单例子。

`IHello.java` 的代码如下：

```

public class HelloSpeaker implements Ihello{
    public void hello(String name){
        System.out.println("Hello, "+name);
    }
}

```

写一个测试程序，如果要使用 `LogHandler` 的 `bind()`方法来绑定被代理类，则 `ProxyDemo.java` 的代码如下：

```

public class ProxyDemo{
    public static void main(String[] args){
        LogHandler logHandler = new LogHandler();
        Ihello helloProxy = (Ihello)logHandler.bind(new HelloSpeaker());
        helloProxy.hello("Justin");
    }
}

```

`HelloSpeaker` 本身的职责是显示文字，却必须插入日志动作，这使得 `HelloSpeaker` 的职责加重。日志的程序代码横切（`cross-cutting`）到 `HelloSpeaker` 的程序执行流程中，日志这样的动作在 `AOP` 中被称为横切关注点（`cross-cutting concerns`）。

使用代理类将记录与业务逻辑无关的动作提取出来，设计为一个服务类，如同前面的范例 `HelloProxy` 或者 `LogHandler`，这样的类称之为切面（`aspect`）。

`AOP` 中的 `aspect` 所指的可以是像日志等的动作或服务，将这些动作（`cross-cutting concern`）设计为通用，不介入特定业务类的一个职责清楚的 `Aspect` 类，这就是所谓的 `Aspect-oriented programming`（`AOP`，面向方面的编程）。

3. AOP术语与概念

1) Cross-cutting concerns (横切关注点)

在前面 `DynamicProxyDemo` 的例子中，记录的动作原先被横切（Cross-cutting）到 `HelloSpeaker` 本身所负责的业务流程中。另外，类似于日志这类的动作，如安全检查、事务等服务，在一个应用程序中常被安排到各个类的处理流程中。这些动作在 AOP 的术语中被称为 Cross-cutting concerns（横切关注点）。

原来的业务流程是很单纯的。Cross-cutting concerns 如果直接写在负责某业务的类的流程中，将使维护程序的成本增加。如果以后要修改类的记录功能或者移除这些服务，则必须修改所有撰写曾记录服务的程序，然后重新编译。另一方面，Cross-cutting concerns 混杂在业务逻辑之中，使得业务类本身的逻辑或者程序的撰写更为复杂。为了加入日志与安全检查等服务，类的程序代码中被硬生生地写入了相关的 Logging、Security 程序片段，加入各种服务的业务流程。

AOP 的核心思想就是将应用程序中的业务逻辑处理部分同对其提供支持的通用服务，即所谓的“横切关注点”进行分离。这些“横切关注点”贯穿了程序中的多个纵向模块的需求。

所谓的这种分离关注就是将某一通用的需求功能从不相关的类中分离出来。同时，能够使得很多类共享一个行为，一旦行为发生变化，不必修改很多类，只要修改这个行为即可。AOP 就是这种实现分散关注的编程方法，它将“关注”封装在“方面”中。

就“关注点”来说，可以将其分为两类：核心关注和横切关注。其中，核心关注点主要关注系统的业务逻辑（如银行管理系统的取款等）；横切关注点主要关注系统级的服务（如日志、事务、安全等），供业务逻辑使用。

2) Aspect（方面）

将散落在各个业务类中的 Cross-cutting concerns（横切关注点）收集起来，设计各个独立可重用的类，这种类称之为 Aspect（方面）。例如，在动态代理中将日志的动作设计为一个 `LogHandler` 类，`LogHandler` 类在 AOP 术语中就是 Aspect 的一个具体实例。在需要该服务的时候，缝合到应用程序中；不需要服务的时候，也可以马上从应用程序中脱离。应用程序中的可重用组件不用做任何修改，例如，在动态代理中的 `HelloSpeaker` 所代表的角色就是应用程序中可重用的组件，在它需要日志服务时并不用修改本身的程序代码。

另一方面，应用程序中可重用的组件以 AOP 的设计方式设计，它不用知道处理提供服务的类的存在，即与服务相关的 API 不会出现在可重用的应用组件中，因而可提高这些组件的重用性，可以将这些组件应用到其他的应用程序中，而不会因为目前加入了某个服务或与目前的应用框架发生耦合。

不同的 AOP 框架对 AOP 概念有不同的实现方式，主要的差别在于所提供的 Aspects 的丰富程度，以及它们如何被缝合（Weave）到应用程序中。

4. 通知Advice

Spring 提供了 5 种通知（Advice）类型：Interception Around、Before、After Returning、Throw 和 Introduction，分别在以下情况被调用。

- Interception Around Advice: 在目标对象的方法执行前后被调用。
- Before Advice: 在目标对象的方法执行前被调用。

- **After Returning Advice:** 在目标对象的方法执行后被调用。
- **Throw Advice:** 在目标对象的方法抛出异常时被调用。
- **Introduction Advice:** 一种特殊类型的拦截通知，只有在目标对象的方法调用完毕后执行。

这里，用前置通知 **Before Advice** 来说明。

Before Advice 会在目标对象的方法执行之前被呼叫。如同在便利店里，在客户买东西之前，老板要给他们一个热情的招呼。为了实现这一点，需要扩展 **MethodBeforeAdvice** 接口。这个接口提供获取目标的方法、参数以及目标对象。

```
public interface MethodBeforeAdvice{
    void before(Method method,Object[] args,Object target)throws Throwable
}
```

【例 5-60】使用 Before Advice。

首先要定义目标对象必须实现的接口。**IHello.java** 的代码如下：

```
public interface IHello{
    public void hello(String name);
}
```

接着定义 **HelloSpeaker**，让它实现 **IHello** 接口。

HelloSpeaker.java 的代码如下：

```
public class HelloSpeaker implements IHello{
    public void hello(String name){
        System.out.println("Hello, "+name);
    }
}
```

在对 **HelloSpeaker** 不进行任何修改的情况下，想要在 **hello()**方法执行之前记录一些信息。有一个组件，但是没有源代码，想对它增加一些日志的服务，此时可以先实现 **MethodBeforeAdvice** 接口。

LogBeforeAdvice.java 的代码如下：

```
import java.lang.reflect.Method;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.springframework.aop.MethodBeforeAdvice;

public class LogBeforeAdvice implements MethodBeforeAdvice{
    private Logger logger = Logger.getLogger(this.getClass().getName());

    public void before(Method method,Object[] args,Object target)throws
        Throwable{
        logger.log(Level.INFO, "method starts..." +method);
    }
}
```

在 **before()**方法中，加入了一些记录信息的程序代码。**LogBeforeAdvice** 类被设计为一个独立的服务，接着在定义文档中定义 **beans-config.xml** 的代码如下：

```
...
<beans>
    <bean id= "logBeforeAdvice" class= "LogBeforeAdvice"/>
```



```

<bean id= "helloSpeaker"class= "HelloSpeaker"/>
<bean id= "helloProxy"class= "org.springframework.aop.framework.
    ProxyFactoryBean">
    <property name= "proxyInterfaces">
        <value>IHello</value>
    </property>
    <property name = "target">
        <ref bean= "helloSpeaker"/>
    </property>
    <property name= "interceptorNames">
        <list>
            <value>logBeforeAdvice</value>
        </list>
    </property>
</bean>
</beans>

```

注意，除了建立 Advice 和 Target 实例之外，还使用了 org.springframework.aop.framework.ProxyBean。这个类会被 BeanFactory 或者 ApplicationContext 用来建立代理对象。需要在“proxyInterfaces”属性中告诉代理可运行的界面，在“target”上告诉 Target 对象，在“interceptorNames”上告诉要应用的 Advice 实例，在不指定目标方法的时候，Before Advice 会被缝合（Weave）到界面上多处有定义的方法之前。

写一个程序测试一下 Before Advice 的动作。SpringAOPDemo.java 代码如下：

```

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public class SpringAOPDemo{
    public static void main(String[] args){
        ApplicationContext context=new FileSystemXmlApplicationContext("bean-
            config.xml");
        IHello helloProxy = (IHello)context.getBean("helloProxy");
        HelloProxy.hello("Justin");
    }
}

```

HelloSpeaker 与 LogBeforeAdvice 是两个独立的类。对于 HelloSpeader 来说，它不用知道 LogBeforeAdvice 的存在，而 LogBeforeAdvice 也可以运行到其他类上。HelloSpeaker 与 LogBefore 都可以重复使用。

5. 切入点PointCut

PointCut 定义了通知 Advice 应用的时机。从下一个实例开始，介绍如何使用 Spring 提供的 org.springframework.aop.support.NameMatchMethodPointcutAdvisor，可以指定 Advice 所要应用的目标上的方法名称，或者用*来指定。例如，hello*表示调用代理对象上以 hello 作为开头的方法名称时，都会应用指定的 Advices。

【例 5-61】 切入点 PointCut。

IHello.java 的代码如下:

```
public interface IHello{
    public void helloNewbie(String name);
    public void helloMaster(String name);
}
```

HelloSpeaker 类实现 IHello 接口。HelloSpeaker.java 代码如下:

```
public class HelloSpeaker implements IHello {
    public void helloNewbie(String name){
        System.out.println("Hello, "+name+ "newbie! ");
    }
    public void helloMaster(String name){
        System.out.println("Hello, "+name+ "master! ");
    }
}
```

编写一个简单的 Advice, 这里使用 Before Advice 中的 LogBeforeAdvice。

定义 Bean 文档, 使用 NameMatchMethodPointcutAdvisor 将 Pointcut 与 Advice 结合在一起。bean-config.xml 的代码如下:

```
...
<beans>
    <bean id= "logBeforeAdvice" class= "LogBeforeAdvice"/>
    <bean id= "helloAdvisor"
        class= "org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
        <property name = "mappedName">
            <value>hello*</value>
        </property>
        <property name= "advice">
            <ref bean= "logBeforeAdvice"/>
        </property>
    </bean>
    <bean id= "helloSpeader" class= "HelloSpeaker"/>
    <bean id= "helloProxy" class= "org.springframework.aop.framework.
        ProxyFactoryBean">
        <property name= "proxyInterfaces">
            <value>IHello</value>
        </property>
        <property name= "target">
            <ref bean= "helloSpeaker"/>
        </property>
        <property name= "interceptorNames">
            <list>
                <value>helloAdvisor</value>
            </list>
        </property>
    </bean>
```

```
</bean>
</beans>
```

在 `NameMatchMethodPointcutAdvisor` 的 “`mappedName`” 属性上，由于指定了 “`hello*`”，所以当调用 `helloNewbie()` 或者 `helloMaster()` 方法时，由于方法名称的开头符合 “`hello`”，就会应用 `logBeforeAdvice` 的服务逻辑，可以写程序来测试。

`SpringAOPDemo.java` 代码如下：

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public class SpringAOPDemo{
    public static void main(String[] args){
        ApplicationContext context=newFileSystemXmlApplicationContext("bean-
            config.
            xml");
        IHello helloProxy= (Ihlllo)context.getBean("helloProxy");
        helloProxy.helloNewbie("Justin");
        helloProxy.helloMaster("Tom");
    }
}
```

在 Spring 中使用 `PointCutAdvisor` 把 `PointCut` 与 `Advice` 结合为一个对象。Spring 中大部分内建的 `PointCut` 都有对应的 `PointAdvisor`。 `Org.springframework.aop.support.NameMatchMethodPointcutAdvisor` 是最简单的 `PointAdvisor`，它是 Spring 中静态的 `PointCut` 实例。使用 `org.springframework.aop.support.RegexpMethodPointcut` 可以实现静态切入点，`RegexpMethodPointcut` 是一个通用的正则表达式切入点，它是通过 `Jakarta ORO` 来实现的。

静态切入点只限于给定的方法和目标类，而不考虑方法的参数。动态切入点与静态切入点的区别是，动态切入点不仅限于给定的方法和类，还可以指定方法的参数。当切入点需要在执行时根据参数值来调用通知时，就需要使用动态切入点。在大多数的切入点可以使用静态切入点，很少有机会创建动态切入点。

6. Spring对事务的支持

事务的特性之一是原子（`Atomic`）性。例如，对数据库存取，就是一组 `SQL` 指令，这组 `SQL` 指令必须全部执行成功；如果因为某种原因（如其中一行 `SQL` 有错误）指令没有执行成功，则先前所执行过的 `SQL` 指令撤销。

在 `JDBC` 中，可以用 `Connection` 的 `setAutoCommit()` 方法，给定它 `false` 参数。在一连串的 `SQL` 语句后面，调用 `Connection` 的 `commit()` 来送出变更。如果中间发生错误，则调用 `rollback()` 来撤销所有的执行。

```
try{
    connection.setAutoCommit(false);
    ...                               //一连串 SQL 操作
    connection.commit();              //执行成功，提交所有变更
}catch(SQLException e) {
    connection.rollback();             //发生错误，撤销所有变更
}
```

在 Spring 中对 `JDBC` 的事务管理加以封装，Spring 事务管理的抽象关键在于

org.springframework.transaction.PlatformTransactionManager 接口的实现。PlatformTransactionManager 接口有许多事务实现类别，如 DataSourceTransactionManager、HibernateTransactionManager、JdoTransactionManager、JtaTransactionManager 等。借助 PlatformTransactionManager 接口和各种技术实现，Spring 在事务管理上可以让开发人员使用一致的编程模式。

事务的失败通常是致命的错误，Spring 不强迫一定要处理，而让开发者自行选择是否要捕捉异常。

Spring 提供编程式的事务管理（Programmatic transaction management）与声明式的事务管理（Declarative transaction management）。

（1）编程式的事务管理。编程式的事务管理可以清楚地控制事务的边界，即自行实现事务何时开始、撤销、结束等，可以实现细粒度的事务控制。

（2）声明式的事务管理。在多数情况下，事务并不需要细粒度的控制，采用声明式的事务管理，优点是 Spring 事务管理的相关 API 可以不用介入程序中，从对象的角度来看，并不知道它正被纳入事务管理中。不需要事务管理的时候，只要在设定档案上修改一些设定，就可以移除事务管理服务。

这里，主要介绍声明式事务管理。Spring 的声明式事务管理依赖 AOP 框架来完成。使用声明式事务管理的好处是事务管理不侵入开发组件，即 DAO 组件不会意识到正处于事务管理之中。如果想要改变事务管理策略，只需要在定义文档中重新组态即可。

例如，可以在不修改 UserDao 类的情况下，为这个类加入事务管理的服务。简化的方法是使用 TransactionProxyFactoryBean，指定要介入的事务管理对象机器方法，这里需要修改定义文档。

【例 5-62】事务管理的 bean-config.xml。

...

```
<beans>
  <bean id= "transactionManager"
    class= "org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name = "sessionFactory">
      <ref bean= "sessionFactory"/>
    </property>
  </bean>
  <bean id= "userDAO" class= "UserDAO"/>
    <property name= "sessionFactory">
      <ref bean= "sessionFactory"/>
    </property>
  </bean>
  <bean id= "userDAOProxy"
    class= "org.springframework.transaction.interceptor.
      TransactionProxyFactoryBean">
    <property name = "proxyInterfaces">
<list>
  <value>IUserDAO</value>
</list>
    </property>
```

```

        <property name= "target">
<ref bean= "transactionManager"/>
</property>
<property name= "transactionAttributes">
    <props>
        <prop key= "insert*">PROPAGATION_REQUIRED</prop>
    </props>
</property>
</bean>
</beans>

```

TransactionProxyFactoryBean 需要一个 TransactionManager，如果是 JDBC，可以使用 DataSourceTransactionManager。由于这里使用的是 Hibernate，所以使用 org.springframework.orm.hibernate3.HibernateTransactionManager。TransactionProxyFactoryBean 是代理类，“target”属性指定要代理的对象，事务管理会自动介入指定的方法前后。这里是指“transactionAttributes”属性指定，insert*表示指定方法名称 insert 开头的全部纳入事务管理。也可以指定方法全名，如果在方法执行过程中发生错误，则所有先前的操作自动撤回，否则正常提交。

insert*等方法指定“PROPAGATION_REQUIRED”，表示在目前的事务执行操作中，如果事务不存在就创建一个新的，相关的意义可以在 API 文件中 TransactionDefinition 接口中找到，可以加上多个事务定义，中间使用逗号“,”隔开。例如，可以加上只读，或者指定某个例外发生时撤回操作：

```
PROPAGATION_REQUIRED, readOnly, -MyCheckedException
```

MyCheckedException 前面加上“-”时，表示发生指定异常撤销操作，如果加上“+”，表示发生异常时立即提交。

由于 userDao 被 userDaoProxy 代理了，所以要做的是取得 userDaoProxy，而不是 userDao。

【例 5-63】SpringAOPDemo.java。

```

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public static void man(String[] args) {
    ApplicationContext context = new FileSystemXmlApplicationContext("beans-
        config.xml");
    User user=new User();
    user.setName("Tome");
    user.setAge(new Integer(20));
    IUserDAO userDao= (IUserDAO)context.getBean("userDAOProxy");
    userDao.insert(user);
}

```

也可以设定更多的事务管理细节。如果以后不再需要事务管理，则直接在 Bean 定义文档中修改配置即可，修改程序进行重新编译等动作。

5.4.4 Spring整合Hibernate

本节讲解 Spring 如何整合 Hibernate 应用。用 Spring 作为该系统框架的构建者，用 Hibernate 作为系统的持久层，同时在 Spring 采用 Bean 来管理持久层和管理层，与 Hibernate 连接进行数据库的操作。Spring 和 Hibernate 天生互补，它们可以在一起很好地协同工作。

1. 在Spring中整合Hibernate的简单实例

本例将演示如何配置 Spring 和 Hibernate，在数据库中插入一条记录。

(1) 创建数据库。先在 MySQL 5.0 中创建 huxidb 数据库，并在其中创建 teacher，其代码如下：

```
CREATE TABLE `teacher` (  
  'id'      varchar(32) collate gb2312_bin NOT NULL default ' ',  
  'number'  varchar(10) character set gb2312 default '0',  
  'name'    varchar(10) character set gb2312 default NULL,  
  'email'   varchar(20) collate gb2312_bin default NULL,  
  'department' varchar(20) character set gb2312 default NULL,  
  'wps'     varchar(16) collate gb2312_bin default NULL,  
  PRIMARY KEY ('id')  
) ENGINE = InnoDB DEFAULT CHARSET = gb2312 COLLATE = gb2312_bin;
```

(2) 在 Eclipse 3.2 中创建 Java 工程 SpringHibernateTestex，在其中添加 src 文件夹，并添加支持 Spring 和数据库操作的包，如图 5-51 所示。

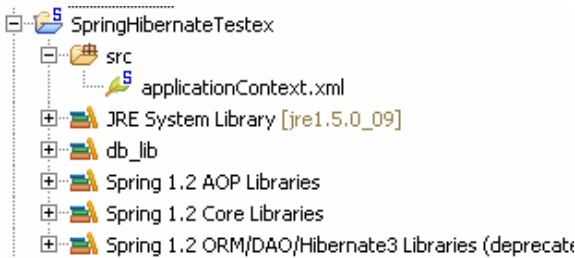


图 5-51 支持 Spring 和数据库的包

(3) 添加支持 Hibernate 的包，并进一步配置文件。用鼠标右键单击工程名，在弹出的快捷菜单中选择 MyEclipse Libraries，然后继续选择 add Hibernate Capabilities 命令，如图 5-52 所示。

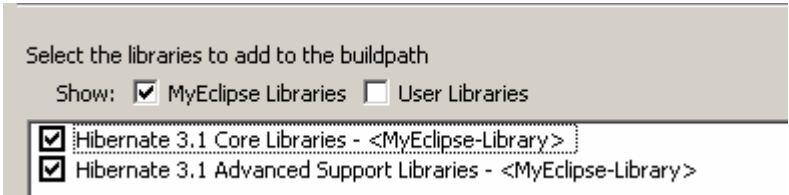


图 5-52 支持 Hibernate 的包

进一步选择 Hibernate 的配置文件，如图 5-53 所示。

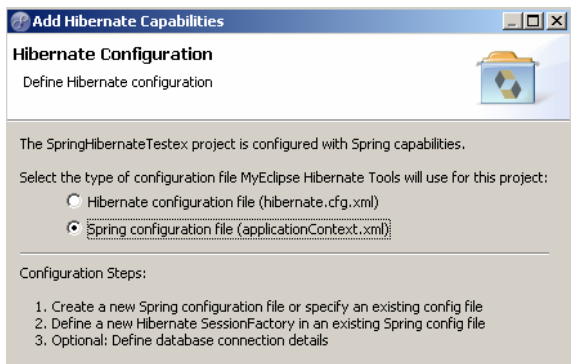


图 5-53 选择 Hibernate 的配置文件

选择 Existing Spring Configuration file，同时填入 sessionFactory 作为 sessionFactory id。在如图 5-54 所示的窗口中，填入相应的信息。

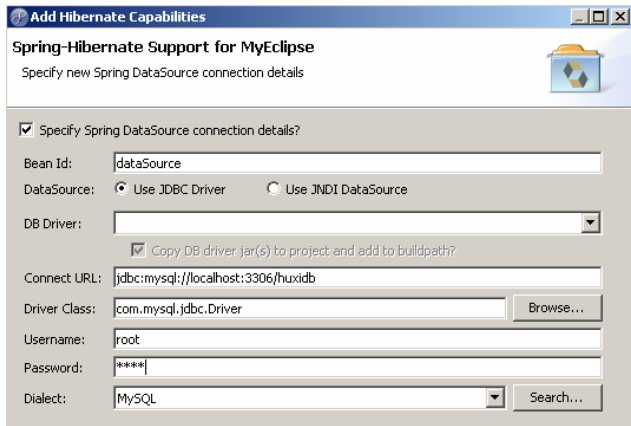


图 5-54 配置 applicationContext.xml

创建并选择 liufy.spring.test 包，并同时生成 HibernateSession Factory.java。生成的 applicationContext.xml 的内容如下：

```
<? xml version = "1.0" encoding = "UTF-8"?>
<! DOCTYPE bean PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-bean.dtd">
<bean>
<bean id = "dataSource"
class = "org.apache.commons.dbcp.BasicDataSource">
<property name = "driverClassName">
<value>com.mysql.jdbc.Driver</value>
</property>
<property name = "url">
<value>jdbc:mysql://localhost:3306/huxidb</value>
</property>
<property name = "username">
<value>root</value>
</property>
<property name = "password">
```

```

        <value>root</value>
    </property>
</bean>
<bean id = "sessionFactory"
class = "org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name = "dataSource">
        <ref bean = "dataSource">
    </property>
    <property name = "hibernateProperties">
        <props>
            <prop key = "hibernatae.dialect">
                org.hibernate.dialect.MySQLDialect
            </prop>
        </props>
    </property>
</bean>
</bean>

```

(4) 在包 `liufy.spring.test` 下创建 `teacher` 表的 POJO 和映射文件。使用 `Hibernate Synchronizer` 工具，将生成的映射文件的 id 修改为下面的形式并存盘。

```

<id name = "Id" type = "string" unsaved-value = "null">
    <column name = "id" not-null = "true"/>
    <generator class = "uuid.hex"/>
</id>

```

具体步骤这里不再叙述。结果如图 5-55 所示。

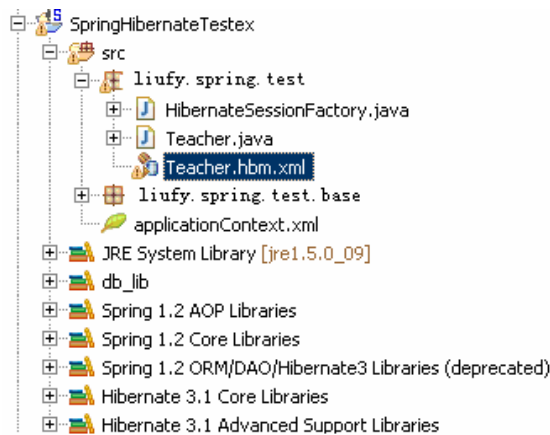


图 5-55 POJO 及映射文件目录结构

(5) 接下来将生成的映射文件添加到配置文件中，在 `sessionFactory` 中修改，内容如下：

```

<bean id = "sessionFactory"
class = "org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name = "dataSource">
        <ref local = "dataSource"/>
    </property>
    <property name = "mappingResources">

```



```

        <list>
            <value>liufy/spring/test/Teacher.hbm.xml</value>
        </list>
    </property>
<property name = "hibernateProperties">
    <props>
        <prop key = "hibernate.dialect">
            org.hibernate.dialect.MySQLDialect
        </prop>
        <prop key = "hibernate.show_sql">true</prop>
    </props>
</property>
</bean>

```

(6) 接下来加入一个叫 `TeacherManager` 类的 DAO (Data Access Object) 来进行增删查改的操作, 代码如下:

```

package liufy.spring.test;

import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
public class TeacherManager extends HibernateDaoSupport {
    public TeacherManager() {
        super();
    }
    public Teacher getTeacher(String name) throws Exception {
        Teacher t = new Teacher();
        t.setName(name);
        java.util.List list =
            this.getHibernateTemplate().findByExample(t);
        if (list.isEmpty())
            throw new Exception("No Such Record");
        else
            return( Teacher) list.get (0);
    }
    public void add(Teacher s) {
        super.getHibernateTemplate().save(s);
        //插入一条数据只需要这一行代码
    }
    public void updateTeacher(Teacher t) {
        this.getHibernateTemplate().update(t);
    }

    public void deleteTeacher(Teacher t) {
        this.getHibernateTemplate().delete(t);
    }
}

```

该类只演示了如何增加、删除、更新、查找一个 `Teacher`, `HibernateTemplate` 还封装了

很多有用的方法，可查阅 **Spring** 文档。**TeacherManager** 中的 **sessionFactory** 是由 **Spring** 注入的，但是 **TeacherManager** 并没有对 **sessionFactory** 做任何的处理。这是因为所有的处理都被 **Hibernate DaoSupport.getHibernateTemplate()**封装。整个 **TeacherManager** 中看不到任何的异常处理，它们也都被基类封装了。

(7) 在 **Spring** 中注册 **teacherManger**，然后向它注入 **sessionFactory**，内容如下：

```
<bean id = "teacherManager" class = "liufy.spring.test.StudentManager">
    <property name = "sessionFactory">
        <ref bean = "sessionFactory"/>
    </property>
</bean>
```

(8) 最后编写测试程序，内容如下：

```
package liufy.spring.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
    FileSystemXmlApplicationContext;

public class MainTest {
    public static void main(String srgs[ ]) {
        try
        {
            ApplicationContext context = new
FileSystemXmlApplicationContext("src/applicationContext.xml");

            Teacher t = new Teacher();
            t.setNaem("李朝容");
            t.setWps("000000");
            System.out.println();
            ( (TeacherManager)context.getBean("teacherManager") ).add(t);

            System.out.println(((TeacherManager)context.getBean("teacherManager"))
.getTeacher("李朝容").toString() );
        }catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

程序显示结果如下：

```
Hibernate:insert into teacher (number,name,email,department,wps,id)values
    (?, ?, ?, ?, ?, ?)
Hibernate:select this_.id as id0_,this_.number as number0_0_,this_.name
    as name0_0_,this
_.email as email0_0_,this_.department as department0_0_,this_.wps as
    wps0_0_from
teacher this_where(this_.name = ?)
```

liufy.spring.test.Teacher@1fb70db7

本测试程序也可以采用 Junit 来进行单元测试。

Spring 已经将 Hibernate 的操作简化到了非常高的程度，最关键的是整个开发可以由设计来驱动。如果一个团队对 Spring 有足够的熟悉，那么完全可以由设计师规划所有的类，整理清楚类之间的关系，写成配置文件，然后编写 Hibernate 映射文件，将数据表与 POJO 关联，成员就可以完全在设计方案内工作。利用 Spring 封装好的 Hibernate 模板，开发起来速度非常快，调试也很容易。

2. Spring整合Hibernate的注意事项

1) 在 Spring 的 Application Context 中创建 SessionFactory

为了避免硬编码造成的资源查找与应用程序对象紧密耦合，Spring 允许用户在 Application Context 中以 Bean 的方式定义，诸如 JDBC DataSource 或者 Hibernate SessionFactory 的数据访问资源。任何需要进行资源访问的应用程序对象只需要持有这些事先定义好的实例的引用即可。

【例 5-64】 创建一个 JDBC DataSource 和 Hibernate SessionFactory。

```
<bean>
<bean id = "myDataSource" class = "org.apache.commons.
dbcp.BasicDataSource"
destroy-method = "close">
    <property name = "driverClassName" value = "org.hsqldb.jdbcDriver"/>
    <property name = "url" value = "jdbc:hsqldb:hsqldb://localhost:9001"/>
    <property name = "username" value = "sa"/>
    <property name = "password" value = "sa"/>
</bean>

<bean id = "sessionFactory"
class = "org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name = "dataSource" ref = "myDataSource"/>
    <property name = "mappingResources">
        <list>
            <value>product.hbm.xml</value>
        </list>
    </property>
    <property name = "hibernateProperties">
        <value>
            hibernate.dialect = org.hibernate.dialect.MySQLDialect
        </value>
    </property>
</bean>
...
</bean>
```

2) HibernateTemplate

HibernateTemplate 提供持久层访问模板化，使用 HibernateTemplate 无须实现特定接

口，它只需要提供一个 `SessionFactory` 的引用，就可以执行持久化操作。`SessionFactory` 对象可通过构造参数传入，或通过设置方式传入。内容如下：

```
//获取 Spring 上下文
ApplicationContext ctx = new FileSystemXmlApplicationContext("bean.xml");
//通过上下文获得 SessionFactory
SessionFactory sessionFactory = (SessionFactory) ctx.getBean
    ("sessionFactory");
```

然后创建 `HibernateTemplate` 实例。`HibernateTemplate` 提供如下 3 个构造函数：

- `HibernateTemplate()`：构造一个默认的 `HibernateTemplate` 实例。因此，使用 `HibernateTemplate` 实例之前，还必须使用方法 `setSessionFactory (SessionFactory sessionFactory)` 来作为 `HibernateTemplate` 传入 `SessionFactory` 的引用。
- `HibernateTemplate (org.hibernate.SessionFactory sessionFactory)`：在构造时已经传入 `SessionFactory` 引用。
- `HibernateTemplate (org.hibernate.SessionFactory sessionFactory, Boolean allowCreate)`：其 `boolean` 型参数表明，如果当前线程已经存在一个非事务性的 `Session`，是否直接返回此非事务性的 `Session`。

对于在 Web 上的应用，通常启动时自动加载 `ApplicationContext`、`SessionFactory` 和 DAO 对象，且这些对象都处在 Spring 上下文管理下，因此，无须在代码中显式设置 `HibernateTemplate` 实例。

`HibernateTemplate` 提供非常多的常用方法来完成基本的操作，例如，常用的增加、删除、修改和查询等操作，Spring 2.0 更增加对命名 SQL 查询的支持，也增加对分页的支持。大部分情况下使用 `Hibernate` 的常规用法，就可完成大多数 DAO 对象的 CRUD 操作。下面是 `HibernateTemplate` 的常用方法。

`void delete (Object entity)`：删除指定持久化实例。

`deleteAll (Collection entities)`：删除集合内全部持久化类实例。

`find (String queryString)`：根据 HQL 查询字符串来返回实例集合。

`findByNameQuery (String queryName)`：根据命名查询返回实例集合。

`get (Class entityClass, Serializable id)`：根据主键加载特定持久化类的实例。

`save (Object entity)`：保存新的实例。

`saveOrUpdate (Object entity)`：根据实例状态，选择保存或者更新。

`update (Object entity)`：更新实例的状态，要求 `entity` 是持久状态。

`setMaxResults (int maxResults)`：设置分页的大小。

【例 5-65】 一个完整 DAO 类的源代码。

```
public class TeacherDAOHibernate implements PersonDAO {
    //采用 log4j 来完成调试时的日志功能
    private static Log log = LogFactory.getLog(NewsDAOHibernate.class);
    private SessionFactory sessionFactory;
    private HibernateTemplate hibernateTemplate = null;
    //设置注入 SessionFactory 必需的 setter 方法
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
}
```

```

//初始化本 DAO 所需的 HibernateTemplate
public HibernateTemplate getHibernateTemplate() {
    //首先, 检查原来的 HibernateTemplate 实例是否还存在
    if(hibernateTemplate == null)
    {
        //如果不存在, 新建一个 HibernateTemplate 实例
        hibernateTemplate = new HibernateTemplate(sessionFactory);
    }
    return hibernateTemplate;
}

//返回全部的人的实例
public List getTeachers()    {
    //通过 HibernateTemplate 的 find 方法返回 Teacher 的全部实例
    return getHibernateTemplate().find("from Teacher");
}

public News getNews(int personid)    {
    return(Person)getHibernateTemplate().get(Teacher.class,new Integer
        (personid) );
}

public void savePerson(Teacher teacher)    {
    getHibernateTemplate().saveOrUpdate(teacher);
}

public void removePerson(int id)    {
    //先加载特定实例
    Object p = getHibernateTemplate().load(Teacher.class,new Integer(id) );
    //删除特定实例
    getHibernateTemplate().delete(p);
}
}

```

HibernateTemplate 还提供了一种更加灵活的方式来操作数据库, 通过这种方式可以完全使用 Hibernate 的操作方式。HibernateTemplate 的灵活访问方式是通过两个方法完成的: Object execute (HibernateCallback action) 和 List execute (HibernateCallback action)。

这两个方法都需要一个 HibernateCallback 的实例, HibernateCallback 实例可在任何有效的 Hibernate 数据访问中使用。程序开发者通过 HibernateCallback, 可以完全使用 Hibernate 灵活的方式来访问数据库, 解决 Spring 封装 Hibernate 后灵活性不足的缺陷。

3) DAO 的实现

在 Spring 中使用 Hibernate, 用户可以使用 Spring 提供的 HibernateTemplate 来实现 DAO 的解决方案。另外, 用户也可以采用其他的方案, 例如, 采用 Hibernate 的原生 DAO 的代码。

【例 5-66】基于原生的 Hibernate API 的 DAO 实现。

```

public class TeacherDaoImpl implements TeacherDao {
    private SessionFactory sessionFactory;
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
}

```

```
public Collection loadTeachersByCategory(String category) {
    return this.sessionFactory.getCurrentSession()
        .createQuery("from test.Teacher t where t.category = ? ")
        .setParameter(0,category)
        .list();
}
}
```

5.5 开发Struts2、Hibernate、Spring集成程序

1. 概述

在简要地了解了 Struts2、Hibernate、Spring 三种框架后，将三种框架集成，开发一个用户登录程序，效果如图 5-56 所示。如果登录成功，则进入欢迎页面。

工程最后的目录树如图 5-57 所示。

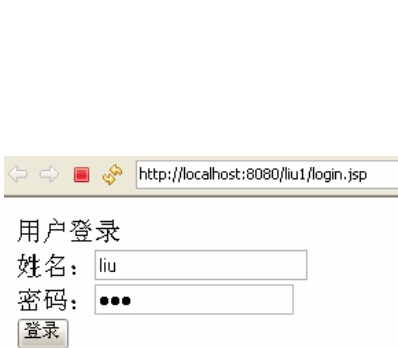


图 5-56 用户登录界面

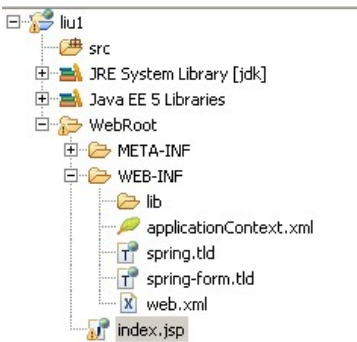


图 5-57 项目目录树

本系统用 Spring 作为该系统框架的构建者，用 Hibernate 作为本系统的持久层，同时在 Spring 中采用 Bean 来管理持久层和管理层，与 Hibernate 连接进行数据库的操作。另外，采用 Struts2 作为系统的表现层。

主要的开发步骤如下：

- (1) 创建数据库表。
- (2) 创建 Web Project。
- (3) 加载用户自定义包和建立 Spring 配置文件。
- (4) 加载 Hibernate 框架。
- (5) 修改 web.xml。
- (6) 增加 struts.properties 文件。
- (7) 反向工程。
- (8) 创建视图层。
- (9) 创建 Action。
- (10) 配置 Spring。
- (11) 部署测试。

2. 具体步骤

1) 创建数据库

如果数据库已经存在，则不需要创建。创建数据库代码如下：

```
CREATE TABLE user(  
    id int(10) not null auto_increment,  
    username varchar(10) not null,  
    password varchar(10) not null,  
    primary key(id)  
)ENGINE=InnoDB DEFAULT CHARSET=GBK;
```

2) 创建 Web Project

用 MyEclipse 下建立新的 Web Project，命名为 S2SH。

3) 加载用户自定义包和建立 Spring 配置文件

用鼠标右键单击工程名，在弹出的快捷菜单中选择 MyEclipse→Add Spring Capabilites，选择 “User Libraries”，如图 5-58 所示，添加用户自定义包 Hibernate3 和配置文件 applicationContext.xml。

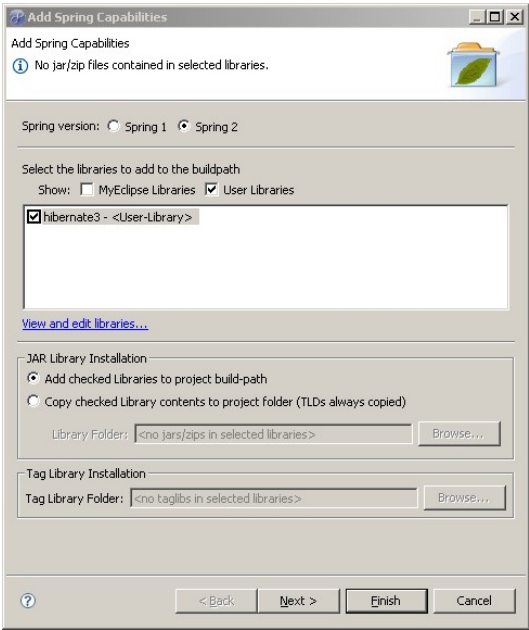


图 5-58 添加 Spring 库

单击 “Next” 按钮，提示是否建立 Spring 配置文件，选择 Spring 配置文件 applicationContext.xml 的地点，默认情况下在 WEB-INF 目录下寻找，如图 5-59 所示。选择默认值，单击 “Finish” 按钮。

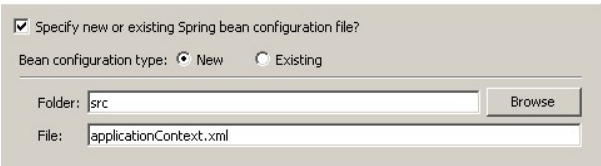


图 5-59 生成 Spring 配置文件

4) 加载 Hibernate 框架

(1) 用鼠标右键单击工程名，在弹出的快捷菜单中选择 **MyEclipse**→**Add Hibernate Capabilities**，选择“**User Libraries**”，添加 jar 包，如图 5-60 所示。

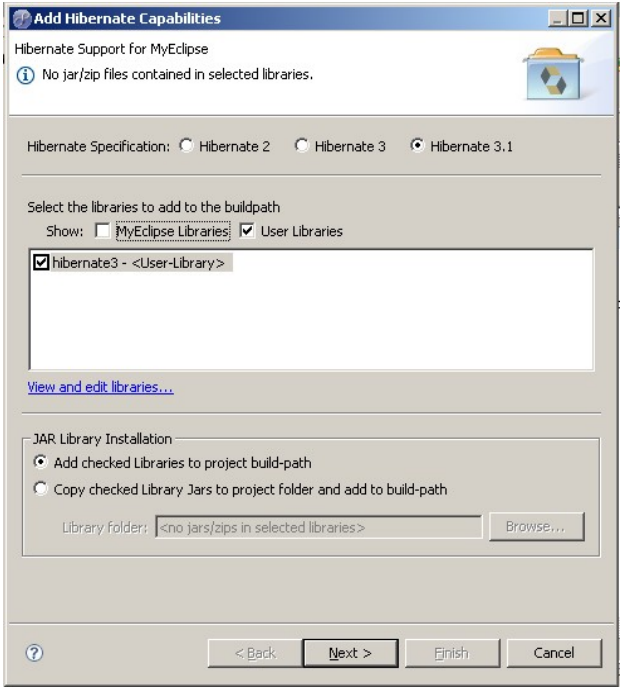


图 5-60 添加 Hibernate 库

(2) 单击“**Next**”按钮，出现如图 5-61 所示的对话框，提示是用 **Hibernate** 的配置文件还是用 **Spring** 的配置文件进行 **SessionFactory** 的配置，选择使用 **Spring** 来对 **Hibernate** 进行管理，这样最后生成的工程就不包含 **hibernate.cfg.xml**，其优势是在一个地方就可以对 **Hibernate** 进行管理了。在 **applicationContext.xml** 中配置连接信息，如图 5-61 所示。

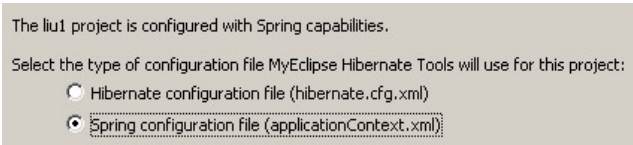


图 5-61 Hibernate 配置文件

(3) 单击“**Next**”按钮，出现如图 5-62 所示的对话框，提示是创建一个新的 **Hibernate** 配置文件还是使用已有的配置文件，由于刚才已经生成了 **Spring** 配置文件，并且要在其中进行 **Hibernate** 的配置，所以选择复选框“**Existing Spring configuration file**”，选择该选项后，下方的“**Spring Config**”后的下拉列表自动填入刚才生成的 **Spring** 配置文件路径，要求填写 **SessionFactory ID**，这个 ID 就是为 **Hibernate** 注入的一个新的 ID，随便起一个名字，如“**sessionFactory**”，单击“**Next**”按钮。为 **SessionFactory** 起一个 ID，如图 5-62 所示。

(4) 单击“**Next**”按钮，出现如图 5-63 所示的对话框，要求选择数据库连接信息。这里需要注意，“**Bean Id**”处填写一个数据源的名称，如“**datasource**”，指定 **Spring** 数据源连接的详细信息，如图 5-63 所示。

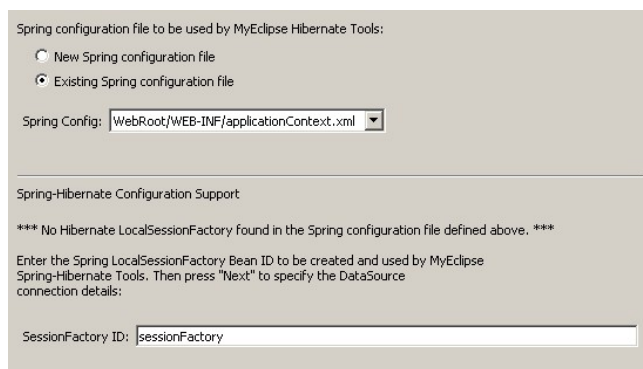


图 5-62 填写 SessionFactory ID

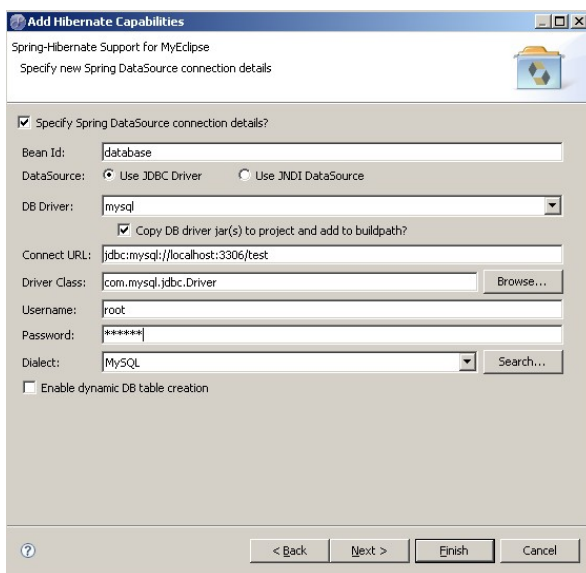


图 5-63 指定 Spring 数据源连接信息

(5) 单击“Next”按钮，出现对话框，提示是否创建“SessionFactory”类，由于本程序 Spring 为注入 sessionFactory，所以不用创建，单击“Finish”按钮。

5) 修改 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <filter>
        <filter-name>struts2</filter-name>
<filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

```

<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/classes/applicationContext.xml
  </param-value>
</context-param>
</web-app>

```

Listener 是 Servlet 的监听器，它可以监听客户端的请求、服务器的操作等。通过监听器，可以自动激发一些操作，如监听到在线的数量。当增加一个 HttpSession 时，就激发了 sessionCreated 方法。

监听器需要知道 applicationContext.xml 配置文件的位置，通过节点<context-param>来配置。

6) 增加 struts.properties 文件

在 src 目录下增加消息包文件 struts.properties，使得 Struts2 的类的生成交给 Spring 完成。步骤：用鼠标右键单击项目的“src”目录，在弹出的快捷菜单中选择 New→File，之后在“Enter or select the parent folder”中填入“struts2_spring/src”，在 File name 栏中写入“struts.properties”，单击“确定”按钮。文件代码如下：

```
struts.objectFactory=spring
```

7) 反向工程

打开 MyEclipse 的 database explorer perspective，右击选择 user 表，选择 Hibernate reverse engineering，生成与数据库表对应的 Java 对象和映射文件，如图 5-64 所示。

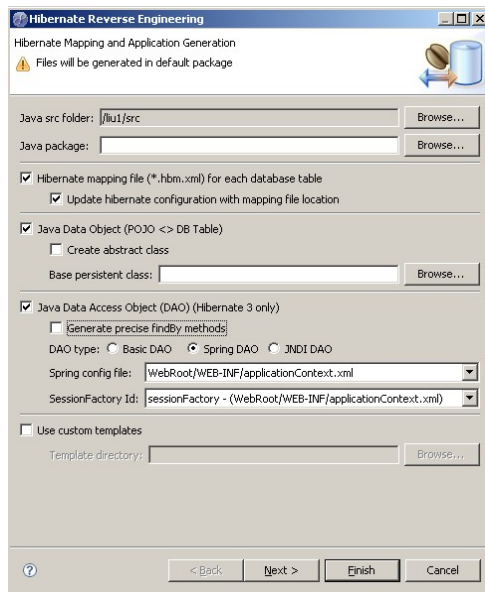


图 5-64 反向工程：Hibernate 映射文件和 POJO 类

单击“Next”按钮，在弹出的对话框中选择 ID 的生成方式为 native，单击“Finish”按钮。

8) 创建视图层

登录页面 login.jsp，代码如下：

```
<%@page language= "java"pageEncoding= "gb2312"%>
<html>
    <head><title>登录页面</title></head>
    <body>
        <form action= "login.action"method= "post">
            用户登录<br>
            姓名: <input type= "text"name= "username"/><br>
            密码: <input type= "text"name= "password"/><br>
            <input type= "submit"value= "登录"/>
        </form>
    </body>
</html>
```

登录成功页面，login_success.jsp，代码如下：

```
<%@page contentType="text/html;charset= "gb2312"%>
<%@taglib prefix="s"uri= "/struts-tags"%>
<html>
    <body>
        <h2>您好! 用户<s:property value= "username"/>欢迎您登录成功</h2>
    </body>
</html>
```

登录失败页面，login_error.jsp，代码如下：

```
<%@page contentType= "text/html;charset=gb2312"%>
<%@taglib prefix= "s"uri= "/struts-tags"%>
<html>
    <body>
        <h2>登录失败</h2>
    </body>
</html>
```

9) 创建 Action

代码如下：

```
import java.util.List;
import org.hibernate.Query;
import org.hibernate.SessionFactory;
import org.hibernate.classic.Session;
import com.opensymphony.xwork2.ActionSupport;
public class LoginAction extends ActionSupport{
    private String username;
    private String password;
    private SessionFactory sessionFactory;
    public String getUsername(){
```

```

        trturn username;
    }
    public void setUsername(String username){
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public SessionFactory getSessionFactory() {
        return sessionFactory;
    }
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
    public String execute() throws Exception{
        Session session= sessionFactory.openSession();
        String hql= "from User u where u.username= ? and u.password=?";
        Query query=session.createQuery(hql);
        query.setParameter(0,username);
        query.setParameter(1,password);
        List user=query.list();
        session.close();
        if(user.size()>0){
            return SUCCESS;
        }else{
            return ERROR;
        }
    }
}

```

配置 struts.xml, 代码如下:

```

<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <package name= "struts"extends= "struts-default">
        <action name="login"class="login">
            <result name= "error">/login_error.jsp</result>
            <result name= "success">/login_success.jsp</result>
        </action>
    </package>
</struts>

```

10) 配置 Spring

代码如下:

```
<?xml version= "1.0" encoding= "UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<beans>
    <bean id= "dataSource" class= "org.apache.commons.dbcp.BasicDataSource">
        <property name= "driverClassName" value= "com.mysql.jdbc.Driver"/>
        <property name= "url" value= "jdbc:mysql://localhost:3306/test">
            </property>
            <property name= "username" value= "root"></property>
            <property name= "password" value= "root"></property>
        </bean>
    <bean id= "sessionFactory" class= "org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
        <property name= "dataSource">
            <ref bean= "dataSource"/>
        </property>
        <property name= "hibernateProperties">
            <props>
                <prop key= "hibernate.dialect">
                    org.hibernate.dialect.MySQLDialect
                </prop>
            </props>
        </property>
        <property name= "mappingResources">
            <list>
                <value>./User.hbm.xml</value>
            </list>
        </property>
    </bean>
    <bean id= "login" class= "LoginAction">
        <property name= "sessionFactory">
            <ref bean= "sessionFactory"/>
        </property>
    </bean>
</beans>
```

在这个典型的应用 **Hibernate** 的程序中, 系统读取配置文件, 并用它来创建 **SessionFactory**。一个 **SessionFactory** 将作用于应用的整个生命期, 用 **SessionFactory** 来获得 **Session** 对象。有了 **Session** 对象, 就能够访问数据库。在应用的整个生命周期, 只要保存一个 **SessionFactory** 实例即可。所以在 **Spring** 配置文件中配置这个对象, 可以用 **Spring** 的 **LocalSessionFactory** 类。

在上例中, **applicationContext.xml** 中有这样一段:

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
```

这个 SessionFactory 需要知道连接到哪个数据库，因此将 DataSource 放入 LocalSessionFactoryBean。

11) 部署测试

选择工具栏中的按钮“Deploy MyEclipse J2EE Project to Server...”将新建的 Web 项目部署到服务器 Tomcat 中。选择项目为 liu1，单击“Add”按钮，选择 Tomcat 6.x 作为服务器，单击“OK”按钮。可用 Junit 进行单元测试。启动 Tomcat，运行程序，在浏览器输入：<http://localhost:8080/liu1/login.jsp>。

习 题 5

1. 什么是 Struts1 和 Struts2?
2. 简述 Struts 的主要结构。
3. 简述如何搭建应用 Struts2 的开发环境。
4. Struts 有哪些核心组件?
5. Struts 的 ActionServlet 和 RequestProcessor 各有什么作用?
6. 登录 Struts 网站，了解 Struts 的最新进展。
7. 什么是 ORM? 请列出一些常见的 ORM 框架。
8. 简述 Hibernate 操作数据库的过程。
9. 编写 Hibernate 更新数据库记录的程序，并调试成功。
10. 编写 Hibernate 删除数据库记录的程序，并调试成功。
11. 简述 Hibernate Synchronizer 的特点。
12. 什么是 Spring? 简述 Spring 的分层结构的思想。
13. 什么是 Spring 的 IoC? 目前有什么实现方式?
14. 按照书中的案例实现简单的 IoC。
15. 什么是 AOP 的切入点?
16. 什么是 AOP 的通知? 它有哪些类型?
17. 如何实现 Spring 的代理?
18. 简述 Spring 的 Bean 配置原则。
19. 利用本章的知识编写实现加锁、解锁的 AOP 实例。
20. 什么是事务? 在 Spring 框架中如何实现事务管理?
21. 将用户注册模块改用 Spring 声明式事务管理实现。
22. 请读者按照本章中 Spring 整合 Hibernate 的例子实现 teacher 对象的插入、删除和更新。

第 6 章 EJB 技术

Java EE 技术之所以赢得企业广泛重视，原因之一就是支持分布式计算的 EJB 技术，提供了一个框架来开发和实施分布式商业逻辑，简化了具有可伸缩性和高度复杂的企业级应用的开发。EJB 规范定义了 EJB 组件及其如何与 EJB 容器进行交互作用。

EJB 主要包含 3 种：会话 Bean、消息驱动 Bean 和实体 Bean。其中，实体 Bean 提供了对象关系映射能力（本章主要介绍单表实体 Bean，多表实体 Bean 关系映射见第 7 章）。但是和 Hibernate 比较又稍显复杂，故在非分布式应用中逊于 Hibernate。随着 Hibernate 在数据持久化方面的影响力逐渐增大，促进了 EJB 3.0 标准的制定，重大变化之一就是简化了 EJB 的开发工作。通过 JDK 5 的新功能 `@Annotation`（翻译为注解或者注释）的使用，减少了开发人员需要提供的文件。目前，Hibernate 和 JDO 都被统一到了 EJB 3.0 框架下，共同支持新的 EJB 3.0 标准。

本章还介绍了 EJB 的工作原理、工作环境及运行，持久化实体管理等，EJB 分布式应用的完整实例请见第 10 章。

6.1 企业级 JavaBean（EJB）：Java EE 解决方案及其特点

应用程序往往分解成多个部分，并分布到一组主机之上。分布式的功能组件以某种方式进行组合，从而为各个网络客户或终端用户提供一个完整的应用。建立分布式的多层应用在设计上有许多好处：这种应用允许自动升级（即服务器端可以在需要时随时调整），而且还支持模块化和逻辑设计。

提供分布式的应用逻辑的 Java EE 解决方案的核心是企业级 JavaBean（EJB）。如前所述，一个 Java EE Web 应用可能包括 4 种不同类型的容器：应用客户容器、Applet 容器、Web 容器以及 EJB 容器。EJB 容器及其管理的对象是我们所关注的重点。一个 EJB 容器包括一个或多个 EJB，其中包含有应用的核心业务逻辑。

Java Servlet 基本上都与 Java EE Web 容器相关，与此不同，EJB 为业务逻辑提供了一个更为灵活的放置位置，且与表示无关。它们可以通过 Servlet、Applet 容器或者 Java 消息服务（JMS）来直接建立联系。而另一方面，Servlet 则是基于 HTML 的瘦客户会话管理的主要方法，而且可以用于向应用分发查询和结果，其形式将基于交互式用户（如 HTML）进行调整。基于 Web 的会话管理和表示交给 Servlet 和 JSP 等技术完成，使得 EJB 可以更专注于业务逻辑的可扩展处理。处理的可扩展性不仅仅针对基于 HTML 的交互式用户，而且也可针对于企业应用集成系统，这些系统可能用于实现批处理或者是一些其他的非交互式的应用访问。

EJB 包含了对多种底层技术的内置支持，这些技术可以提高服务器端应用的可扩展性，具体包括以下几方面。

（1）对象保存：EJB 可以自动与持久性存储器进行集成，从而使应用更具健壮性和可分

布性。这为开发人员提供了两种选择：自行负责定义对象保存的详细情况，或者将所有工作都交给容器完成。

(2) 事务管理：开发人员所编写的用于实现事务管理的代码通常都是不一样的，而且有时还会导致令人不满意的性能。一般在分布式对象之间设置代码来协调事务很困难，而且也容易出错。为了避免这种手工实现事务管理的方式所带来的问题，可以提供一种平台级的服务，从而自动地完成这一管理过程。在针对 EJB 的 Java EE 规范中，提供了对分布式事务管理的内置支持。充分利用这一特点可以让开发人员编写出更简单、更准确无误的 EJB，而且不用担心在分布式环境中何时管理事务，以及如何进行管理。

(3) 位置透明性：无论 EJB 客户位于网络上的哪个位置，找到 EJB 并远程执行其功能都是很简单的，而且效率也很高。这种功能可以简单地实现应用功能的分布和重新部署，它可以获得更好的可扩展性。

除了这种显式的或“明显”的 EJB 特性之外，规范中还提供了一些其他的特性，这些特性则更为隐式或“隐蔽”（但同样非常重要）。这些特性对于性能和可扩展性有着重要的影响。本章将在稍后加以介绍，现在首先学习一些相关的基本知识，从而了解 EJB 的工作原理及开发方法。

6.2 EJB的工作原理、环境及运行

6.2.1 EJB的工作原理及类型

1. EJB工作原理

在 Java EE 模型中，EJB 是由容器所管理的分布式对象。而实际的工作则由各个 Bean 实例来完成。容器提供了可以代表客户与这些实例进行交互的代理（EJB 对象）。代理要负责根据需要进行 Bean 的创建与撤销，换句话说，要管理相应 Bean 实例的生存周期。客户、EJB 容器和 Bean 实例之间的关系如图 6-1 所示。

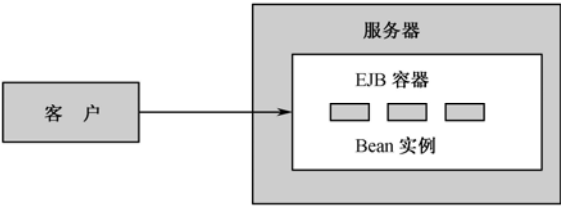


图 6-1 EJB 客户、容器和 Bean 实例之间的交互

在 J2EE 的 EJB 2.1 标准中，客户与一个 EJBObject 通信，这是由 EJB 容器所提供的。实际上有两种类型的 EJB 对象接口，将分别对它们做简要的介绍。EJBObject 在客户与 Bean 之间的通信中相当于一个中间人，对 Bean 实例的分配要由容器来调整。

在客户/EJB 通信中实际涉及两种 EJBObject。为了更好地理解它们，需要对 EJB 如何使用做更为具体的说明。与应用无关，客户与 EJB 的交互包括以下步骤：

- (1) 客户得到一个 EJBObject 的句柄。
- (2) 客户根据需要调用此对象的业务方法。

(3) 在使用之后，客户将放弃 EJBObject 的句柄。

在此需要两种类型的 EJBObject 接口：对于第 1 步和第 3 步需要一个主（home）接口，而对于第 2 步则需要一个本地（local）或远程（remote）接口。主接口的作用是为所请求的 EJB 提供类似于工厂的服务（即创建和撤销方法）。对于同一类型的 EJB，主接口还提供了一种“寻找” Bean 实例的方法。本地或远程接口的使用是为调用 Bean 所封装的应用逻辑（即业务方法）提供一种简单的 API。本地接口和远程接口之间的差别应该已经很明确了：前者表示由位于 Bean 所在主机上的客户进行访问，而后者则表示由不在同一主机上的客户进行访问。

容器将通过每个 Bean 实例的生命周期方法来管理 Bean，在此将它们作为功能黑箱来处理。这些方法是与 Bean 创建、撤销、激活和挂起等操作相关的回调型方法。一般地，由容器将一个 Bean 转换为某种特定状态时，就会调用一个或多个此类方法。下面将对每一种方法进行定义和讨论。

(1) 创建：如果存在着客户需求，但是又没有可用的实例时，就需要创建 Bean 实例。这样，容器必须实例化一个新的实例。

(2) 撤销：Bean 实例可以得到周期性的垃圾回收或清除。

(3) 激活：Bean 实例可以是一个实例池中的成员，这样在一个的新的客户请求到来时，容器就可以从此池中获得 Bean 实例。它对性能有着显著的影响，因为对请求分配一个实例池中的成员比创建一个新实例再分配给请求要廉价得多。

(4) 挂起：正如激活是实例创建的一种“简易（lite）”方式，挂起也可以理解为实例撤销的一个“简易”方式。这里在使用后不是将实例删除，而是将 bean 实例返回给实例池。这说明，在需要时该实例还可以重新激活。

对于容器和 EJB 的工作以及它们如何合作有了初步的了解，下面再来看看各种不同的 EJB 类型其应用条件。

2. EJB类型

目前在 EJB 规范中支持三种基本的 Bean 类型。

(1) 会话 Bean：它与一个特定的业务操作相关，特别是与一个交互性会话中所请求的操作有关。例如，合计订单的逻辑就可以放在会话 Bean 中。顾名思义，会话 Bean 是一种主要用于同步、交互性会话的应用接口。

(2) 实体 Bean：它与一个需要持久保存的应用对象有关。例如，订单和客户对象本身都可以分别用实体 Bean 来表示。在需要管理持久性数据时，会话 Bean 和消息驱动 Bean 通常会在执行期间与实体 Bean 进行交互。

(3) 消息驱动 Bean：它与一个特定的业务操作相关，特别是在应用集成或周期性批处理中所需要的操作。例如，所有存储的订单可能要在每个工作日之后通过一个消息驱动 Bean 进行批处理。消息驱动 Bean 能通过 JMS 来访问。

图 6-2 显示了不同 EJB 类型之间的一般关系。注意，有些客户通过会话 Bean 调用应用，而有些客户则通过消息 Bean 调用应用。无论怎样，这些“接口型” Bean 可以相互交互，而且还可能与表示持久性数据的一组实体 Bean 实现交互。

使用 EJB 的大型应用中往往会涉及多种 EJB 类型。

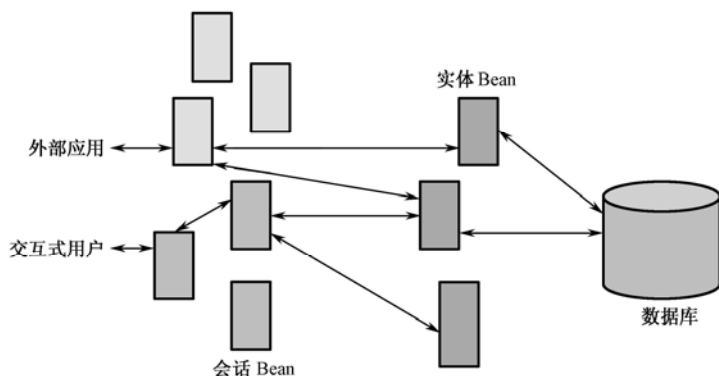


图 6-2 各类 EJB 的作用及可能的 Bean 间的关系

6.2.2 EJB 3.0 的特点及运行实例

1. EJB 3.0 的新特点

EJB 2.1 标准让开发者付出了很多辛苦，特别是当把 EJB 和 Hibernate 相比较的时候，实体 Bean 显得很复杂，EJB 3.0 可以说是 EJB 标准的一次简化运动，去除了很多复杂的细节，减轻了开发者的负担，提高了开发效率。

EJB 3.0 的主要目的就是简化 EJB 的开发，体现在下面几点。

(1) 定义 Java 语言元数据注释，简化了开发者的工作，减少了开发人员需要提供的文件，不需要对 EJB 的部署进行描述，有关 EJB 的信息可以从 EJB 的类中直接获得并自动生成接口。

(2) 不需要开发人员维护 EJB 容器的通用过程，只需要进行一些配置。

(3) 靠注解来封装 EJB 的工作环境、JNDI 服务和简化的查找机制。

(4) 简化 EJB 类型，使用简单的 POJO 型的 JavaBean。

(5) 会话 Bean 和实体 Bean 不再需要 EJB 组件接口（EJBObject、EJBLocalObject 和 Remote Interface）。会话 Bean 的业务接口是一个普通的 Java 接口。实体 Bean 不再需要任何接口。

(6) EJB 对象不再需要 Home 接口。

(7) CMP 类型的实体 Bean 被简化。

(8) 支持轻量级的继承和多态模型。

(9) 利用 Java 语言的元数据，通过注解来实现实体 Bean 的对象—关系映射。

(10) 加强 EJB QL 来提供更强的可用性，包括内连接、外连接、批量更新和删除、子查询、group by、动态查询和本地 SQL 语句等。

(11) 不再需要 Exception 接口。

(12) 不再需要实现 Callback 接口。

(13) 增强了 EJB 对象的测试方法，提供了在容器外测试 EJB 对象的能力，提高了开发效率。

2. 运行环境配置

目前支持 Java EE 1.4 标准的开源应用服务器主要有两种，一种是 JBoss，另外一种 Apache Geronimo。JBoss 出现得比较早，应用比较广泛，本书以 JBoss 为开发环境介绍 EJB

的开发和使用。Geronimo 是 IBM 捐赠的一个开源软件，来源于 Websphere 的社区版。由于有 IBM 的强力的技术支持和市场推广，其发展情景十分光明。

本书使用 JBoss-4.2.0.GA 来运行 EJB 3.0 组件。开发和运行环境主要包含 JBoss 应用服务器、JBossIDE 和 Eclipse 3.4.1。在安装 JBoss 之前，确认已经安装了 JDK 5.0 或以上版本的 Java 运行环境。

解压缩后，在“系统变量”里添加 JBOSS_HOME 变量，值为 JBoss 的安装路径，如 D:\Java\jboss-4.2.0.GA。现在验证是否安装成功，在 DOS 窗口中执行下面命令：

```
D:\Java\jboss-4.2.0.GA\bin>run -c all
```

这个命令使用完全配置启动 JBoss，如果没有 Java 异常抛出，就表示安装成功了。

JBossIDE 是一系列的 Eclipse 插件，使用这些插件后，就可以在 Eclipse 中开发 EJB 并部署到 JBoss 服务器。

3. 运行一个EJB 3.0 的实例

服务器安装成功，并在 Eclipse 中配置好后，即可以在 Eclipse 中开发一个 HelloWorldEJB（通常第一个程序都是 HelloWorld 程序）部署到 JBoss 服务器上。

【例 6-1】在 Eclipse 中开发一个 HelloWorldEJB，部署到 JBoss 服务器上。

（1）在 Eclipse 中创建一个普通的 Java 项目，名字为 HelloWorldEJB。然后单击“Add External JARs”按钮，把 D:\Java\jboss-4.2.0.GA\client 下面所有的支持包加入到项目的“Build Path”中。单击“Add Library”按钮，把 JUnit 也加入到项目的构建路径中，如图 6-3 所示。

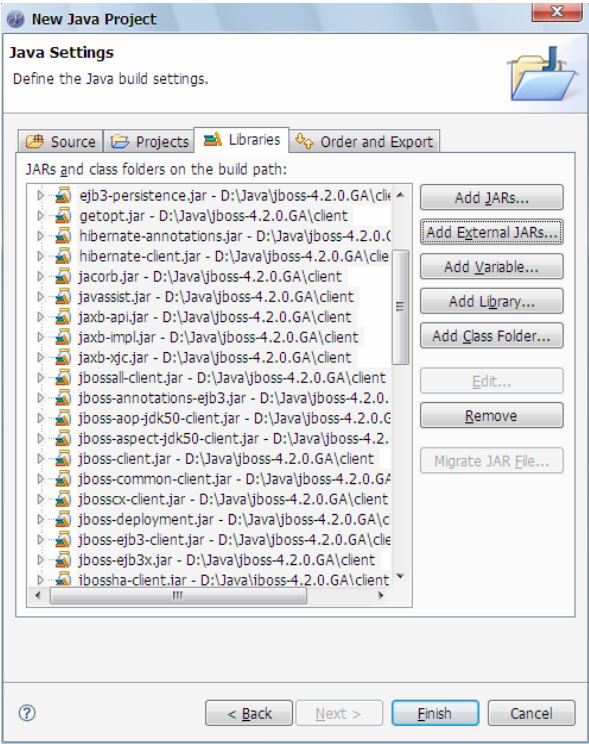


图 6-3 EJB 项目的构建路径设置

（2）在项目中创建两个文件，一个是接口 HelloWorld，另一个是类 HelloWorldBean。

HelloWorldBean 实现接口程序 HelloWorld。HelloWorld 的程序如下：

```
package edu.CEC.ejb3;

public interface HelloWorld {

    public String SayHello(String name);

}
```

HelloWorldBean 的程序如下：

```
package edu.CEC.ejb3.impl;
import edu.CEC.ejb3.HelloWorld;
import javax.ejb.Remote;
import javax.ejb.Stateless;
<!-- 用注解@Stateless 表示这个 EJB 是一个无状态会话 Bean -->
@Stateless
<!-- 用注解@Remote 指明了它的远程接口 -->
@Remote( { HelloWorld.class } )
public class HelloWorldBean implements HelloWorld {

    public String SayHello(String name) {

        return name + ":这是我的第一个 EJB3.";

    }

}
```

(3) 创建一个 EJBFactory 类，用于使用 JNDI 获取 EJB 对象，内容如下：

```
package junit.debug;
import java.util.Properties;
import javax.naming.InitialContext;
import javax.naming.NamingException;
public class EJBFactory {

    public static Object getEJB(String jndipath) {

        try {

            Properties props = new Properties( );
            props.setProperty("java.naming.factory.initial",
                             "org.jnp.interfaces.NamingContextFactory");
            props.setProperty("java.naming.provider.url", "localhost:1099");
            props.setProperty("java.naming.factory.url.pkgs",
                             "org.jboss.naming:org.jnp.interfaces");
            InitialContext ctx = new InitialContext(props);
            return ctx.lookup(jndipath);

        } catch (NamingException e) {

            e.printStackTrace();

        }

        return null;

    }

}
```

(4) 创建一个测试 EJB 类的 JUnit 测试类 HelloWorldTest，内容如下：

```
package junit.debug;
import static org.junit.Assert.*;
import org.junit.BeforeClass;
```

```

import org.junit.Test;
import edu.CEC.ejb3.HelloWorld;
public class HelloWorldTest {
    protected static HelloWorld helloworld;
    @BeforeClass
    public static void setUpBeforeClass( ) throws Exception {
        helloworld=(HelloWorld)EJBFactory.getEJB("HelloWorldBean/remote");
    }
    @Test
    public void testSayHello( ) {
        assertEquals("成工院人: 这是我的第一个 EJB3.",helloworld.SayHello(“成工
院人” ) );
    }
}

```

创建后的目录结构如图 6-4 所示。

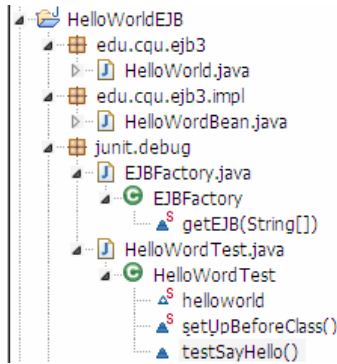


图 6-4 HelloWorldEJB 项目的目录结构

(5) 创建一个 build.xml 文件，编译和发布 HelloWorldEJB 到 JBoss 服务器。build.xml 文件如下：

```

<? xml version = "1.0" encoding = "UTF-8"? >
<!-- -EJB3 HelloWorld build file - -->
<project name = "HelloWorldEJB" default = "ejbjar" basedir = ".">
    <property environment = "env"/>
    <property name = "app.dir" value = "$ { basedir}"/>
    <property name = "src.dir" value = "$ { app.dir}/src"/>
    <property name = "jboss.home" value = "$ { env.JBOSS_HOME}"/>
    <property name = "jboss.server.config" value = "all"/>
    <property name = "build.dir" value = "$ { app.dir}/build"/>
    <property name = "build.classes.dir" value = "${ build.dir}/classes"/>
    <property name = "ejbjar.name" value = "$ { ant.project.name}"/>
    <!-- -Build classpath - -->
    <path id = "build.classpath">
        <fileset dir = "D:\Java\jboss-4.2.0.GA\client">
            <include name = "*.jar"/>
        </fileset>
    </path>

```

```

    <pathelement location = "${build.classes.dir}"/>
</path>
<!-- Prepares the build directory -->
<target name = "prepare" depends = "clean">
    <mkdir dir = "${build.dir}"/>
    <mkdir dir = "${build.classes.dir}"/>
</target>
<!-- Compiles the source code -->
<target name = "compile" depends = "prepare" description = "编译">
    <Javac srcdir = "${src.dir}" destdir = "${build.classes.dir}" debug=
        "on" deprecation = "on" optimize = "off" includes = "edu/* **">
    <classpath refid = "build.classpath"/>
</Javac>
</target>
<target name = "ejbjar" depends = "compile" description = "创建 EJB 发布包">
    <jar jarfile = "${app.dir}/${ejbjar.name}.jar">
    <fileset dir = "${build.classes.dir}">
    <include name = "** */*.class"/>
    <exclude name = "junit/debug/*.*"/>
    </fileset>
</jar>
</target>
<target name = "deploy" depends = "ejbjar">
    <copy file= "${app.dir}/${ejbjar.name}.jar" todir= "${jboss.home}
        /server/${jboss.server.config}
        /deploy"/>
</target>
<!-- Cleans up generated stuff -->
<target name = "clean">
    <delete dir = "${build.dir}"/>
    <delete file = "${jboss.home}/server/${jboss.server.config}
        /deploy/${ejbjar.name}.jar"/>
</target>
</project>

```

(6) 在 Eclipse 中启动 JBoss Server。在 Eclipse 中选择 HelloWorldTest 类，单击鼠标右键弹出快捷菜单，选择“Run As”→“JUnit Test”命令，如图 6-5 所示。

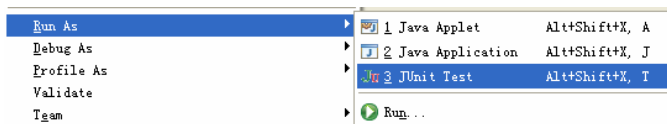


图 6-5 运行 JUnit 测试类

上面的步骤简单地测试了一下 EJB。JBoss 还提供了调试 EJB 的功能，如果需要调试 EJB，那么以“Debug”方式启动服务器，然后在程序中设置断点即可。

6.2.3 独立的Tomcat调用EJB

在正式的生产环境下，大部分 EJB 的客户端所在的服务器都为独立的 Tomcat。在独立的 Tomcat 服务器中调用 EJB 需要执行以下操作步骤。

(1) 根据应用的需要，把调用 EJB 所依赖的 JAR 包复制到 Tomcat 下的 shared/lib 目录或者 Web 应用的 Web-INF/lib 目录下。所依赖的包文件一般在 JBoss 安装目录的 client 目录中，如 D:\Java\jboss-4.2.0.GA\client。还有一些必备文件，如：

- Jbossall-client.jar。
- Jbossall-remoting.jar。
- Jbossall-aop-jdk50-client.jar。
- Jbossall-aspect-jdk50-client.jar。
- Jbossall-ejb3-client.jar。
- Jbossall-ejb3x.jar。
- Jbossall-annotations-ejb3.jar。

(2) 把调用的 EJB 接口复制到应用的\Web-INF\classes\目录下。在此环境下，不能使用 EJB 的 Local 接口，因为它与 JBoss 不在同一个 VM 中。

提示：一般 Web 程序都使用了日志记录组件 Log4j。JBoss 服务器也使用了 Log4j，在所有的类路径中最好不要出现不同版本的 Log4j 组件，并且最好使用较新的版本，这样可以避免或者减少一些错误的发生。

6.2.4 EJB的类和接口

1. EJB的类

在 EJB 3.0 中，EJB 的类是最主要的程序部分。如果 EJB 需要一个业务接口，则 EJB 的类可以实现业务接口，或者从 EJB 的类中生成一个业务接口。

在 EJB 3.0 中，开发一个 EJB 类，开发者可以用 Java 元数据来注解这个类。这些注解可以用来生成其他程序需要的部分，定义 EJB 容器需要的语义，或者提供发布时需要的结构和配置信息等，这些注解是写入代码且被编译到 Class 中的。

EJB 的类必须定义为 Bean 类型或者实现下面的接口：

- Javax.ejb.SessionBean。
- Javax.ejb.EntityBean。
- Javax.ejb.MessageDrivenBean。

在 EJB 3.0 中，没有必要直接去实现这些接口，可以在 EJB 的类中直接加入注解 @Stateless、@Stateful、@MessageDriven 或者 @Entity 来定义一个 EJB 类。

下面的程序就是一个无状态会话 Bean。在程序中用注解 @Stateless 表示这个 EJB 是一个无状态会话 Bean，用注解 @Remote 表示它的远程接口。

【例 6-2】一个无状态会话 Bean 的程序。

```
package edu.CEC.ejb3.impl;
import edu.CEC.ejb3.HelloWorld;
import Javax.ejb.Remote;
```

```
import javax.ejb.Stateless;
@Stateless
@Remote({HelloWorld.class})
public class HelloWorldBean implements HelloWorld {
    public String SayHello(String name) {
        return name + ":这是我的第一个 EJB3.";
    }
}
```

2. EJB的业务接口

在 EJB 3.0 中，EJB 的业务接口就是一个普通的 Java 接口，不再需要 EJB 2.1 版本规定的那些接口了。EJB 的类在实现接口时采用下面的规则。

(1) 一个没有实现任何接口的 EJB 类，将会实现它自己生成的业务接口，这个接口无论是否被注释为 Remote 都将被实现为一个 Local 接口。生成的接口名将是 EJB 的类的名字去掉了 Bean、Impl、EJB 后缀构成的，并遵守命名规则。

(2) 在默认状态下，EJB 的类的所有 public 方法，除了那些因容器需要而被容器调用的方法之外，都将存在于生成的接口中。

(3) 如果 EJB 的类实现了一个单一接口，那么接口将被假设为 Bean 的业务接口。这个接口将被实现为 local 接口，除非接口被注释为 Remote。

(4) 一个 EJB 的类也可以实现多个接口，如果 EJB 的类实现了多个接口，任何 EJB 的类的业务接口必须被明确注释为 @Local 或者 @Remote。

(5) 设定 EJB 的类实现 Web service 相关的接口时，需要依靠 JSP-181 标准。

3. EJB类的变化

关于异常，业务接口的方法可以声明任何应用程序的异常。但是，业务接口的方法不可以抛出 Java.rmi.RemoteException 类型的异常，即使接口是远程接口或者 EJB 的类在被注解为 @Remote 后也不可以。如果在数据通信中发生了问题，容器将抛出 EJBException 异常，这个异常包含 RemoteException 或者系统内的异常。

关于 EJBObject 和 EJBLocalObject，会话 Bean 的业务接口不需要再扩展 EJBObject 或者 EJBLocalObject 接口了，这些接口将被容器实现的接口所扩展。如果一定要访问 EJBObject 或者 EJBLocalObject 接口，就必须强行转换业务接口。

关于 Home 接口，在 EJB 3.0 中，不再需要 Home 接口，客户端通过 JNDI 可以直接获取 EJB 对象。对于实体 Bean，客户端可以通过实体 Bean 的 new() 来创建一个实例。实体 Bean 的实例由 EntityManager 来维护。

6.3 会话Bean

会话 Bean (SessionBean) 与一个特定的业务操作相关，特别是与一个交互性会话中所请求的操作有关。顾名思义，会话 Bean 是一种主要用于同步、交互性会话的应用接口。

会话 EJB 有两种类型：有状态和无状态的类型。

6.3.1 无状态会话Bean

无状态会话 Bean 不会保留请求之间的状态，因此可以用于处理来自任何客户的请求。由于它们不与任何客户相关联，无状态会话 Bean 的数目不必等同于并发会话的数目。会话会经常处于非激活状态，因此可能只需要少数无状态会话 Bean 来处理多个客户处理应用请求。从本质上讲，即从 Bean 的角度看，这种做法比起有状态会话 Bean 而言，将具有更好的可扩展性。

从前面的描述中可以看到，无状态会话 Bean 就相当于一种双赢。对 Bean 建立缓冲池而不是为每个客户建立一个交互 Bean，这样做将会改善可扩展性，同时还可以保留资源。不过，需要记住，会话通常都需要状态管理。例如，在线零售应用往往需要虚拟的购物车，因此必须在某处保存状态。例如，可以在客户端（通过 Cookie）实现，也可以在重写的 URL、服务器端内存，或者数据库中进行保存。为状态管理设计一个有效的解决方案是应用设计人员所面临的一个难题。

前面例 6-2 设计的 HelloWorldEJB 就是一个无状态会话 Bean。

6.3.2 有状态会话Bean

有状态会话 Bean 会在与客户通信时维护状态。更具体地讲，它们会保留客户请求之间的实例变量值。在客户和 Bean 会话结束时（也就是说，客户结束了此会话），这一状态就消失了。很明显，由于 Bean 维护了客户请求之间的状态，因此它对于客户继续与同一个 Bean 通信是相当重要的。从理论上说，有多少并发会话就应该有多少有状态会话 Bean，因为每个会话都有一个状态需要得到管理的客户。根据 Java EE 规范，有状态会话 Bean 可以周期性地写入持久性存储器中。

有状态会话 Bean 的开发与无状态会话 Bean 类似，改变一些元数据的注解就可以将无状态的会话 Bean 变成有状态的会话 Bean。但是对于服务器来说，需要为有状态会话 Bean 维持数据的状态。通过客户端的程序，可以创建一个在服务器中保存的有状态会话 Bean，让会话 Bean 跟踪客户端而存在。

【例 6-3】在 HelloWorldEJB 项目中，创建一个有状态会话 Bean。

(1) 创建一个业务接口，必须继承 Serializable 接口，这样 EJB 容器才能在它们不再使用时序列化存储其状态信息。代码如下：

```
package edu.CEC.ejb3;
import java.io.Serializable;
public interface MyAccount extends Serializable {
    public int Add(int a,int b);
    public int getResult( );
}
```

(2) 创建一个业务类，实现业务接口。代码如下：

```
package edu.CEC.ejb3.impl;
import edu.CEC.ejb3.MyAccount;
import javax.ejb.Remote;
import javax.ejb.Stateless;
@SuppressWarnings("serial")
```

```

@Stateful
@Remote(MyAccount.class)
public class MyAccountBean implements MyAccount {
    private int total = 0;
    private int addresult = 0;
    private int Add(int a,int b) {
        addresult = a +b;
        return addresult;
    }
    public int getResult( ) {
        total + = addresult;
        return total;
    }
}

```

MyAccountBean 直接实现 MyAccount 接口，通过@Stateful 注解定义这是一个有状态会话 Bean，@Remote 注解指明有状态 Bean 的 remote 接口。@SuppressWarnings (“serial”) 注解屏蔽缺少 serialVersionUID 定义的警告。

EJB 创建完成后，重新编译，发布到 JBoss 服务器。

(3) 创建一个测试类 MyAccountTest，内容如下：

```

package junit.debug;
import edu.CEC.ejb3.MyAccount;
public class MyAccountTest {
    public static void main(String[ ] args) {
        MyAccount A = (MyAccount) EJBFactory.getEJB("MyAccountBean/remote");
        System.out.println("调用 A.Add( )的结果是: "+A.Add(1,1) );
        System.out.println("调用 A.getResult( )的结果是: "+A.getResult ( ) );
        System.out.println("-----");
        MyAccount B = (MyAccount)EJBFactory.getEJB("MyAccountBean/remote");
        System.out.printle("调用 B.Add( )的结果是: "+ B.Add(1,1) );
        System.out.println("调用 B.getResult( )的结果是: "+B. getResult ( ) );
    }
}

```

执行该类的输出结果如下：

调用 A.Add()的结果是： 2

调用 A.getResult()的结果是： 2

调用 B.Add()的结果是： 2

调用 B.getResult()的结果： 2

因为有状态会话 Bean 的每个用户都有自己的一个实例，所以两者对有状态会话 Bean 的操作不会影响对方。

如果把 MyAccountBean 修改为无状态会话 Bean，运行结果如下：

调用 A.Add()的结果是： 2

调用 A.getResult()的结果是：2

调用 B.Add()的结果是：2

调用 B.getResult()的结果：4

比较两次运行的结果，可以看出，对于无状态会话 Bean，多个用户共享一个 Bean。

6.4 消息驱动Bean

消息驱动 Bean (MDB, MessageDrivenBean) 是设计用来专门处理基本消息请求的组件。它与一个特定的业务操作相关，特别是在应用集成或周期性批处理中所需要的操作。例如，所有存储的订单可能要在每个工作日之后通过一个消息驱动 Bean 进行批处理。消息驱动 Bean 与会话 Bean 有一定的相似性，因为它们都与一个业务工作有关。不过，消息驱动 Bean 并不用于交互式会话。因此，不需要维护会话调用状态，这样看来它们更像是无状态会话 Bean。另外，类似于无状态会话 Bean，它们可以由多个客户使用，从而比有状态会话 Bean 更具有可扩展性。消息驱动 Bean 所采用的是一种异步监听型的通信。客户通过 JMS 发送消息，而 JMS 则交其转发给消息驱动 Bean，这样就会自动执行一个 Bean 方法 onMessage()。这种异步通信与会话 Bean 和实体 Bean 通信中所涉及的同步形式的通信有所不同，并且更为有效。

一个 MDB 类必须实现 MessageListener 接口。当容器检测到 Bean 守候的队列中的一条消息时，就调用 onMessage()方法，将消息作为参数传入。MDB 在 OnMessage()中决定如何处理该消息。用户可以用注释来配置 MDB 监听哪一条队列。当 MDB 部署时，容器将会用到其中的注解信息。当一个业务执行的时间很长，而执行结果无须实时向用户反馈时，很适合使用消息驱动 Bean。

下面在 HelloWorldEJB 项目中创建一个简单的消息驱动 Bean。其 Bean 的代码逻辑是把接收到的消息打印出来。

【例 6-4】在 HelloWorldEJB 项目中创建一个简单的消息驱动 Bean。

```
package edu.CEC.ejb3;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName= "destinationType",propertyValue
        = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName =
"destination",propertyValue = "queue/ejb3/send") })
public class SendBean implements MessageListener {
    public void onMessage(Message msg) {
        TextMessage tmsg = (TextMessage) msg;
        try {
```

```

        System.out.println(tmsg.getText( ) );
    } catch (JMSEException e) {
        e.printStackTrace( );
    }
}
}

```

上面通过 `@MessageDriven` 注释指明这是一个消息驱动 Bean，并使用 `@ActivationConfigProperty` 注释配置消息的各种属性，其中 `destinationType` 属性指定消息的类型。消息有两种类型：Topics 和 Queues。

Topics 可以有多个客户端。用 Topic 发布允许一对多或多对多通信通道。消息的产生者称做 Publisher，消息接收者称做 Subscriber。使用 `destinationType` 属性对应值为 `Java.jms.Topic`。

Queues 仅仅允许一个消息传送给一个客户。一个发送者将消息放入消息队列，接收者从队列中抽取并得到消息，消息就会在队列中消失。第一个接收者抽取并得到消息后，其他人就不能再得到它。`destinationType` 属性对应值为 `Javax.jms.Queue`。

`destination` 属性用做指定消息路径，消息驱动 Bean 在发布时，如果路径不存在，容器会自动创建该路径。当容器关闭时，该路径会自动被删除。

运行本例子，当一个消息到达 `queue/ejb3/send` 队列时，就会触发 `onMessage` 方法，消息作为一个参数传入，在 `onMessage` 方法里面得到消息体，并把消息内容打印到控制台上。

【例 6-5】访问 SendBean 的代码。

```

package junit.debug;
import Java.util.Properties;
import Javax.jms.Queue;
import Javax.jms.QueueConnection;
import Javax.jms.QueueConnectionFactory;
import Javax.jms.QueueSender;
import Javax.jms.QueueSession;
import Javax.jms.TextMessage;
import Javax.naming.InitialContext;
public class SendBeanTest {
    public static void main(String[] args) throws Exception{
        Properties props = new Properties( );
        QueueConnection cnn = null;
        QueueSender sender = null;
        QueueSession sess = null;
        Queue queue = null;
        props.setProperty("Java.naming.factory.initial",
            "org.jnp.interfaces.NamingContextFactory");
        props.setProperty("Java.naming.provider.url", "localhost:1099");
        props.setProperty("Java.naming.factory.url.pkgs", "org.jboss.naming");
        InitialContext ctx = new InitialContext(props);
        QueueConnectionFactory factory = (QueueConnectionFactory) ctx
            .lookup("ConnectionFactory");
    }
}

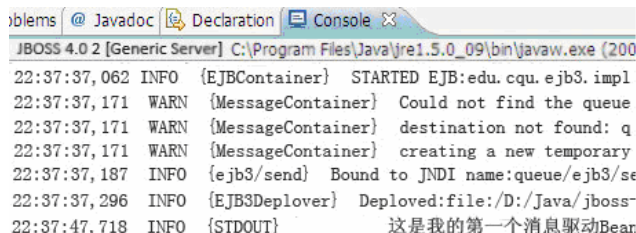
```

```

cnn = factory.createQueueConnection( );
sess = cnn.createQueueSession(false,QueueSession.AUTO_ACKNOWLEDGE);
queue = (Queue)ctx.lookup("queue/ejb3/send");
TextMessage msg = sess.createTextMessage("这是我的第一个消息驱动 Bean");
sender = sess.createSender(queue);
sender.send(msg);
sess.close( );
System.out.println("消息已经发送出去了,你可以到JBoss 控制台查看 Bean 的输出");
}
}

```

在 JBoss 服务器的控制台中可以看到如图 6-6 所示的输出结果。



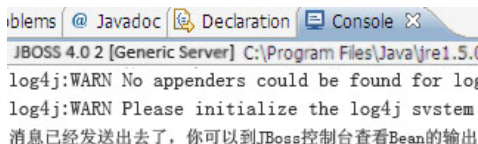
```

JBoss 4.0.2 [Generic Server] C:\Program Files\Java\jre1.5.0_09\bin\javaw.exe (200
22:37:37,062 INFO {EJBContainer} STARTED EJB:edu.cqu.ejb3.impl
22:37:37,171 WARN {MessageContainer} Could not find the queue
22:37:37,171 WARN {MessageContainer} destination not found: q
22:37:37,171 WARN {MessageContainer} creating a new temporary
22:37:37,187 INFO {ejb3/send} Bound to JNDI name:queue/ejb3/se
22:37:37,296 INFO {EJB3Deployer} Deployed:file:/D:/Java/jboss-
22:37:47,718 INFO {STDOUT} 这是我的第一个消息驱动Bean

```

图 6-6 消息队列中的消息

在测试程序的输出控制台中可以看到如图 6-7 所示的输出结果。



```

JBoss 4.0.2 [Generic Server] C:\Program Files\Java\jre1.5.0
log4j:WARN No appenders could be found for log
log4j:WARN Please initialize the log4j system
消息已经发送出去了,你可以到JBoss控制台查看Bean的输出

```

图 6-7 消息 Bean 的控制台输出

测试代码利用 `QueueConnectionFactory` 创建队列消息的连接,然后将 `Text` 类型的消息发送到名为 `queue/ejb3/send` 的队列,服务器端的 `SendBean` 接收到消息而做出响应。

6.5 实体Bean

实体 Bean 与一个需要持久保存的应用对象有关。一个实体 Bean 对应一个持久对象,而不是一个业务工作。例如,订单和客户对象本身都可以分别用实体 Bean 来表示。在需要管理持久性数据时,会话 Bean 和消息驱动 Bean 通常会在执行期间与实体 Bean 进行交互。

在定义一个实体 Bean 时,可指定其持久保存由容器管理还是由 Bean 管理。简单地说,容器管理的持久保存 (Container-Managed Persistence, CMP) 说明 Java EE 开发商会对如何实现实体 Bean 的保存进行处理,而 Bean 管理的持久保存 (Bean-Managed Persistence, BMP) 则意味着必须自己对这种方式进行编码。在 EJB 3.0 中 CMP 类型的实体 Bean 被简化了。

EJB 3.0 中的实体 Bean 变化最大,不仅使用了注解,而且引入了持久化类 POJO 的方式,在某种程度上体现了 Hibernate 的主要特点,使开发实体 EJB 更加简单。

6.5.1 实体Bean配置文件及JBoss的数据源

1. 实体Bean的persistence.xml配置文件

现在 EJB 3.0 实体 Bean 是纯粹的 POJO，可以像开发一般的 JavaBean 一样编程，只需做少量的注解来定义实体关系及 O/R 映射等。

一个实体 Bean 由实体类和 persistence.xml 文件组成。persistence.xml 文件在 JAR 文件的 META-INF 目录。也可以用一个 hibernate.properties 属性文件代替 persistence.xml 文件。

persistence.xml 文件指定实体 Bean 使用的数据源及 EntityManager 对象的默认行为。

【例 6-6】persistence.xml 文件的配置。

```
<persistence>
  <persistence-unit name = "CEC">
    <jta-data-source>Java:/DefaultMySqlDS</jta-data-source>
    <properties>
      <property name = "hibernate.hbm2ddl.auto" value = "create-drop"/>
    </properties>
  </persistence-unit>
</ persistence>
```

persistence-unit 节点可以有一个或多个，每个 persistence-unit 节点定义了持久化内容名称、使用的数据源名称及 Hibernate 属性。name 属性用做设置持久化名称。jta-data-source 节点用做指定实体 Bean 使用的数据源名称。指定数据源名称时 Java: /前缀不能缺少，数据源名称大小写敏感。properties 节点用做指定 Hibernate 的各项属性，如果 hibernate.hbm2ddl.auto 的值设为 create-drop，在实体 Bean 发布及卸载时将自动创建及删除相应数据库表（注意：JBoss 服务器启动或关闭时会引发实体 Bean 的发布及卸载）。Properties 节点的可用属性及默认值可以在 JBoss 安装目录的\server\all\deploy\ejb3.deployer\META-INF\persistence.properties 文件中看见。

如果用户的表已经存在，并且想保留数据，发布实体 Bean 时可以把 hibernate.hbm2ddl.auto 的值设为 none 或 update，以后为了实体 Bean 的改动能反映到数据表，建议使用 update，这样实体 Bean 添加一个属性时，能同时在数据表增加相应字段。本书中用的是已经创建好的数据库表，故使用属性值 update。

2. JBoss数据源的配置

JBoss 有一个默认的数据源 DefaultDS，它使用 JBoss 内置的 HSQLDB 数据库。实际应用中，用户可能使用不同的数据库，如 MySQL、MS SQLServer、Oracle 等。各种数据库的数据源配置模板可以在 JBoss 安装目录的\docs\examples\jca 目录中找到，默认名称为：数据库名+ -ds.xml。

不管使用哪种数据库都需要把它的驱动类 JAR 包放置在 JBoss 安装目录的\server\all\lib 目录下，放置后需要重新启动 JBoss 服务器。

本书中使用的数据库是 MySQL 5。数据源文件配置好后，需要放置在 JBoss 安装目录的\server\config-name\deploy 目录下，本书采用的配置名为：all，所以路径为 JBoss 安装目录的\server\all\deploy 目录。

下面定义一个名为 DefaultMySqlDS 的 MySQL 数据源，连接数据库为 filedb，数据库登录用户名为 root，密码为 password，数据库驱动类为 com.mysql.jdbc.Driver，只须修改数据库名及登录用户名密码就可以直接使用。

【例 6-7】 一个名为 DefaultMySqlDS 的 MySQL 数据源。

```
< ? xml version = "1.0" encoding = "UTF-8"?>
<datasources>
  <local-tx-datasource>
    <jndi-name>DefaultMySqlDS</jndi-name>
    <connection-url>jdbc:mysql://localhost:3306/filedb</connetion-url>
    <driver-class>com.mysql.jdbc.Driver</ driver-class>
    <user-name>root</ user-name>
    <password>password</password>
    <exception-sorter-class-name>
org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter
    </exception-sorter-class-name>
    <metadata>
      <type-mapping>mysql</ type-mapping>
    </metadata>
  </ local-tx-datasource>
</ datasources>
```

6.5.2 单表实体Bean及持久化实体管理器

1. 单表实体Bean

在开发实体 Bean 之前，确保配置数据源放置在 JBoss 安装目录的\server\all\deploy 目录下，把数据库驱动 JAR 包放置在 JBoss 安装目录的\server\all\lib 目录下，放置后需要重启 JBoss 服务器。

这里创建 tblUser 表，见表 6-1。

表 6-1 tblUser 用户表

序 号	字 段	类 型	含 义
1	UserID	Varcher (50) ,not null	主键，用户唯一标识
2	UserName	Varcher (10) ,not null	姓名
3	UserMail	Varcher (50) ,not null	电子邮件
4	UserPassword	Varcher (50) ,not null	口令
5	UserType	Int,default 0	用户类型：0 为普通用户，1 为管理员
6	UserCreated	Datetime default getdate()	创建时间

(1) 建立与 tblUser 表进行映射的实体 Bean。

【例 6-8】 建立与 tblUser 表进行映射的实体 Bean。

```
package edu.CEC.ejb3;
import Java.io.Serializable;
import Java.util.Date;
```

```

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
@SuppressWarnings("serial")
@Entity
@Table(name = "tblUser")
public class User implements Serializable {
    private String userID;
    private String userName;
    private String userMail;
    private String userPassword;
    private Date userCreated;
    private int userType = 0;//普通用户
    @Id
    public String getUserID( ) {
        return userID;
    }
    public void setUserID(String userID) {
        this.userID = userID;
    }
    @Column(nullable = false,length = 50)
    public String getUserMail( ) {
        return userMail;
    }
    public void setUserMail(String userMail) {
        this.userMail = userMail;
    }
    @Column(nullable = false,length = 10)
    public String getUsername( ) {
        return userName;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    @Column(nullable = false,length = 50)
    public String getUserPassword( ) {
        return userPassword;
    }
    public void setUserPassword (String userPassword) {
        this.userPassword = userPassword;
    }
    @Column(nullable = false)
    public int getUserType ( ) {
        return userType;
    }
}

```



```

    }
    public void setUserType(int user Type) {
        this.userType= userType;
    }
    @Column(nullable = false)
    public Date getUserCreated ( ) {
        return userCreated;
    }
    public void setUserCreated (Date user Created) {
        this.userCreated= userCreated;
    }
}

```

通过 User 类可以看到开发实体 Bean 非常简单，就像开发一般的 Java Bean 一样，@Entity 注解指明这是一个实体 Bean，每个实体 Bean 类映射数据库中的一个表，@Table 注解的 name 属性指定映射的数据表名称，User 类映射的数据表为 tblUser。实体 Bean 的每个实例代表数据表中的一行数据，行中的一列对应实例中的一个属性。

① @Column 注解定义了映射到列的所有属性，如列名是否唯一、是否允许为空、是否允许更新等。

- name: 映射的列名。例如，映射 tblUser 表的 userName 列，可以在 name 属性的 getUserNaem 方法上面加入 @Column (name = “username”)，如果不指定映射列名，容器将属性名称作为默认的映射列名。
- unique: 是否唯一。
- nullable: 是否允许为空。
- length: 对于字符型列，length 属性指定列的最大字符长度。
- insertable: 是否允许插入。
- updatable: 是否允许更新。
- columnDefinition: 定义建表时，创建此列的 DDL。
- secondaryTable: 从表名。如果此列不建在主表上（默认建在主表），该属性定义该列所在从表的名字。

关于@Columnm 的详细的说明可以参考在线文档：

<http://Java.sun.com/Javaee/5/docs/api/Javax/persistence/Column.html>

② @Id 注解指定 userID 属性为表的主键，它可以有多种生成方式。

- table: 容器指定用底层的数据表确保唯一。
- sequence: 使用数据库的 sequence 列来保证唯一。
- indentity: 使用数据库的 indentity 列来保证唯一。
- auto: 由容器挑选一个合适的方式来保证唯一。
- none: 容器不负责主键的生成，由调用程序来完成。

本例子的主键是字符串，由用户输入，没有使用自动创建方式。

(2) 创建使用实体 Bean 的会话 Bean。为了使用实体 Bean，定义一个会话 Bean 作为它的使用者。

【例 6-9】创建使用实体 Bean 的会话 Bean。

```

package edu.CEC.ejb3;
import Java.util.Date;

```

```

import Java.util.List;
public interface UserDao {
    public boolean insertUser(String userID,String username,String userMail,
        String userPassword,Date userCreated,int userType);
    public String getUserByNameByID(String userID);
    public boolean updateUser(User user);
    public User getUserByID(String userID);
    public List getUserList( );
}

```

下面是会话 Bean 的实现类，内容如下：

```

package edu.CEC.ejb3.impl;
import Java.util.Date;
import Java.util.List;
import edu.CEC.ejb3.User;
import edu.CEC.ejb3.UserDAO;
import Javax.ejb.Remote;
import Javax.ejb.Stateless;
import Javax.persistence.EntityManager;
import Javax.persistence.PersistenceContext;
import Javax.persistence.Query;
@Stateless
@Remote ( { UserDao.class } )
public class UserDaoBean implements UserDao {
    @PersistenceContext
    protected EntityManager em;
    public User getUserByID(String userID) {
        User user = em.find(User.class,userID);
        return user;
    }
    public List getUserList( ) {
        try {
            Query query = em.createQuery("from User u order by userid asc");
            List list= query.getResultList( );
            em.clear( );
            //分离内存中受 EntityManager 管理的实体 Bean, 让 VM 进行垃圾回收
            return list;
        } catch (Exception e) {
            e.printStackTrace( );
            return null;
        }
    }
    public String getUserByNameByID(String userID) {
        User user = em.find(User.class,userID);
        return user.getUserName( );
    }
    public Boolean insertUser(String userID,String username,String

```

```

        userMail,String userPassword,Date userCreated,int userType) {
try {
    User user = new User( );
    user.setUserID(userID);
    user.setUserName(userName);
    user.setUserMail(userMail);
    user.setUserPassword(userPassword);
    user.setUserCreated(userCreated);
    user.setUserType(userType);
    em.persist(user);
} catch (Exception e) {
    e.printStackTrace( );
    return false;
}
return true;
    }

    public boolean updateUser(User user) {
try {
    em.merge(user);
} catch(Exception e) {
    e.printStackTrace( );
    return false;
}
return true;
    }
}

```

上面使用到了一个对象：EntityManager em。EntityManager 是由 EJB 容器自动地管理和配置的，不需要用户自己创建，它用做操作实体 Bean。

上面 em.find()方法用做查询主键为 userID 的记录。em.persist()方法用做向数据库插入一条记录。大家可能感觉奇怪，在类中并没有看到对 EntityManager em 进行赋值，后面却可以直接使用它。这是因为在实体 Bean 加载时，容器通过@PersistenceContext 注解动态注入 EntityManager 对象。

如果 persistence.xml 文件中配置了多个不同的持久化内容。用户需要指定持久化名称注入 EntityManager 对象，可以通过@PersistenceContext 注解的 unitName 属性进行指定，例如：

```

@PersistenceContext(unitName = "CEC")
EntityManager em;

```

如果只有一个持久化内容配置，则不需要明确指定。

下面是 persistence.xml 文件的配置，内容如下：

```

< ? xml version = "1.0" encoding = "UTF-8"?>
<persistence xmlns = "http://Java.sun.com/xml/ns/persistence"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation = "http://Java.sun.com/xml/ns/persistence
        http://Java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version = "1.0">
    <persistence-unit name = "CEC">

```

```

<jta-data-source>Java:/DefaultMySqlDS</jta-data-source>
<properties>
    <property name= "hibernate.hbm2ddl.auto" vaue = "update"/>
</ properties>
</ persistence-unit>
</ persistence>

```

到目前为止，实体 Bean 应用已经开发完成，开发后的目录结构如图 6-8 所示。

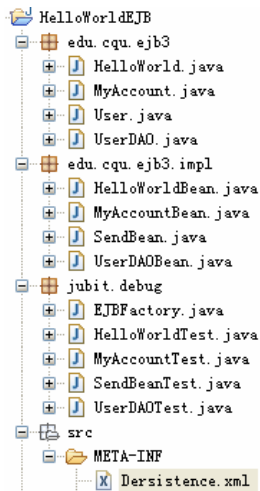


图 6-8 开发完成的实体 Bean 目录结构

(3) 发布 EJB。接下来，把它打包发布到 JBoss 服务器。对前面的 build.xml 文件做一下修改，把 persistence.xml 文件也打包到 JAR 文件中。修改 ejbjar target，增加 META-INF 文件夹。

【例 6-10】 修改 ejbjar target，增加 META-INF 文件夹，创建 EJB 发布包。

```

<target name = "ejbjar" depends = "compile" description = "创建 EJB 发布包">
    <jar jarfile = "${app.dir}/HelloWorldEJB.jar">
        <fileset dir = "${build.classes.dir}">
            <include name= "* */*.class"/>
            <exclude name = "junit/debug/*.*/>
        </fileset>
        <metainf dir = "${src.dir}/META-INF">
            <include name = "* */>
        </metainf>
    </jar>
</target>

```

运行 deploy target 就把打包后的 HelloWorldEJB.jar 文件复制到 JBoss 服务器，服务器会自动完成 EJB 的部署。

(4) 测试 EJB。在 Eclipse 中，使用 JUnit 创建一个 EJB 的测试类。

【例 6-11】 在 Eclipse 中，使用 JUnit 创建一个 EJB 的测试类。

```

package junit.debug;
import static org.junit.Assert.*;
import Java.util.Date;

```

```

import Java.util.List;
import org.junit.BeforeClass;
import org.junit.Test;
import edu.CEC.ejb3.User;
import edu.CEC.ejb3.UserDAO;
public class UserDAOTest {
    private static UserDAO dao;
    @BeforeClass
    public static void setUpBeforeClass( ) throws Exception {
        dao = (UserDAO)EJBFactory.getEJB("UserDAOBean/remote");
    }
    @Test
    public void testInsertUser( ) {
        assertTrue(dao.insertUser("user001", "测试用户 1", "user@CEC.edu.cn",
            "password",new Date( ),0 ) );
    }
    @Test
    public void testGetUserNameByID( ) {
        assertNotNull(dao.getUserNameByID("user001") );
    }
    @Test
    public void testUpdateUser( ) {
        User user = dao.getUserByID("user001");
        user.setUserName("测试用户 2");
        assertNotNull(user);
        assertTrue(dao.updateUser(user) );
    }
    @Test
    public void testGetUserByID( ) {
        assertNotNull(dao.getUserByID("user001") );
    }
    @Test
    public void testGetUserList( ) {
        List list = dao.getUserList( ) {
            assertNotNull(list);
            assertEquals(false,list.isEmpty( ) );
        }
    }
}

```

按照“JUnit Test”方式运行这个测试类，运行结果可以看出 5 个测试方法全部成功。

检查 MySQL 数据库 filedb 中的表 tblUser 的数据，测试数据已经存入数据库，说明前面创建的实体 Bean 和会话 Bean 都能够正确执行。

2. 持久化实体管理器

持久化实体管理器（EntityManager）是用来对实体 Bean 进行操作的辅助类。它可以用来产生或删除持久化的实体 Bean，通过主键查找实体 Bean，也可以通过 EJB 3.0 QL 语言查找满足条件的实体 Bean。实体 Bean 被 EntityManager 管理时，EntityManager 跟踪它的状态

改变，在决定更新实体 Bean 的时候便会把发生改变的值同步到数据库中。当实体 Bean 从 EntityManager 分离后，它是不受管理的，EntityManager 无法跟踪它的任何状态改变。EntityManager 的获取前面已经介绍过，可以通过@PersistenceContext 注解由 EJB 容器动态注入，例如：

```
@PersistenceContext(unitName = "CEC")
```

```
EntityManager em;
```

下面介绍 EntityManager 常用的 API。

(1) find(): 如果知道 Entity 唯一标识符，可以通过 find()方法，获取该实体。例如：

```
User user = em.find(User.class, "user001");
```

(2) persist(): 保存 Entity 到数据库，如 em.persist(user)。

(3) flush(): 当实体正在被容器管理时，用户可以调用实体的 set 方法对数据进行修改，在容器决定调用 flush 时，更新的数据才会同步到数据库。如果用户希望修改后的数据实时同步到数据库，可以执行 EntityManager.flush()方法。

(4) merge(): 在实体 Bean 已经脱离了 EntityManager 的管理时使用，当容器决定调用 flush 时，数据将会同步到数据库中，如 em.merge(user)。

(5) remove(): 把 Entity 从数据库中删除，如 em.remove(user);

(6) createQuery(): 除了使用 find()或 getReference()方法来获得 Entity Bean 之外，还可以通过 EJB 3.0 QL 得到实体 Bean。要执行 EJB 3.0 QL 语句，必须通过 EntityManager 的 createQuery()或 createNamedQuery()方法创建一个 Query 对象。例如：

```
Query query = em.createQuery( "select p from User p where p.userID =  
                                'user001'");
```

```
List result = query.getResultList( );
```

(7) createNativeQuery(): 用于直接执行数据库 SQL 语句查询。例如：

```
Query query = em.createNativeQuery( "select * from tblUser",User.class);
```

```
List result = query.getResultList( );
```

(8) refresh(): 如果用户怀疑当前被管理的实体已经不是数据库中最新的数据，可以通过 refresh()方法刷新实体，容器会把数据库中的新值重写进实体。这种情况一般发生在获取了实体之后，有人更新的数据库中的记录，这时需要得到最新的数据。

(9) clear(): 在处理大量实体的时候，如果用户不把已经处理过的实体从 EntityManager 中分离出来，这将会消耗大量的内存。调用 EntityManager 的 clear()方法后，所有正在被管理的实体将会从持久化内容中分离出来。

习 题 6

1. EJB 3.0 规范，简化了 EJB 的开发，主要体现在哪几个方面？
2. 简述如何下载和安装 JBoss 服务器。
3. JBoss 的各个子目录有什么含义？
4. 如何在 Eclipse 配置中使用 JBoss 服务器的开发环境？
5. Ant 在程序开发过程中有什么作用？
6. 运行在 Tomcat 中的 Web 程序如何调用运行在 JBoss 上的 EJB？
7. 简述在 Eclipse 中，如何开发 EJB 3.0 的实体 Bean。

第 7 章 Java EE持久性数据管理

7.1 Java持久性API简介

Java 持久性 API 为 Java 开发程序员提供了一个对象/关系映射工具，以便管理 Java 应用程序中的相互关联的数据。Java 持久性由以下三个部分组成。

- (1) Java 持久性 API。
- (2) 查询语言。
- (3) 对象/关系映射元数据。

7.1.1 实体

一个实体是一个轻量级的持久性域对象 (domain object)。通常情况下一个实体代表关系数据库中的一个表，而每个实体的实例相当于表中的一行。实体的基本编程工具是实体类，但实体可以使用帮助类。

一个实体的持久状态可以通过持久性字段 (persistent fields) 或持久性属性 (persistent properties) 来体现。这些字段或属性通过使用对象/关系映射的注解，将实体和实体的关联关系映射到底层数据库中的数据和数据之间的关系上。

1. 对实体类的要求

实体类必须符合下列要求。

- (1) 实体类必须用 `javax.persistence.Entity` 注解来注明。
- (2) 实体类必须有一个 `public` 或 `protected` 修饰的无参数的构造方法。实体类还可以有其他的构造方法。
- (3) 实体类不得声明为 `final`。方法或持久性实例变量可以不必声明为 `final`。
- (4) 如果一个实体的实例作为一个游离的对象被按值传递，例如，通过一个会话 bean 的远程业务接口，该类必须实现 `Serializable` 接口。
- (5) 实体可以扩展实体类和非实体类，并且非实体类可以扩展实体类。
- (6) 持久实例变量必须声明为 `private`, `protected`, 或者 `package-private`, 并且只能被实体的方法直接访问。客户必须通过访问器或者业务方法来访问实体的状态。

2. 实体类中的持久性数据成员和成员属性(Persistent Fields and Properties)

实体的持久性状态可通过两种方式访问，一种是通过实体的实例变量来访问，或通过 `JavaBeans-style` 的成员属性来访问。数据成员或成员属性必须是下列 Java 语言的类型：

- (1) Java 基本数据类型。
- (2) `java.lang.String`。
- (3) 其他实现了 `serializable` 接口的类型，包括：

Wrappers of Java primitive types; java.math.BigInteger; java.math.BigDecimal; java.util.Date; java.util.Calendar; java.sql.Date; java.sql.Time; java.sql.Timestamp; User defined serializable types; byte[]; Byte[]; char[]; Character[]).

(4) 枚举类型。

(5) 其他实体和实体的集合。

(6) 可嵌入的类。

实体可以使用持久性数据成员 (fields) 或持久性成员属性 (properties)。如果实体的实例变量应用了映射注解, 实体使用持久性数据成员 (fields); 如果映射注解应用于该实体的 JavaBean-style 成员属性的 getter 方法, 实体使用持久性成员属性 (properties)。不能在一个实体中同时在应用映射注解到数据成员和成员属性。

如果实体类使用持久性数据成员, 持续运行时直接访问实体类的实例变量。所有没有注明 javax.persistence.Transient 或没有标记为 Javatransient 的数据成员将被持久性到数据存储。在对象/关系映射注解必须被应用于实例变量。

如果实体使用持久性成员属性, 该实体必须遵循 JavaBeans 组件的方法的约定。JavaBeans-style 的成员属性使用 getter 和 setter 方法, 这些方法通常在实体类的实例变量的名称之后命名。对于每一个实体的持久性成员属性的类型, 由 getter 方法获取成员属性和 setter 方法设置成员属性。如果成员属性是一个布尔值, 可以使用 is Property 而不是 get Property。例如, 如果一个 Customer 实体使用永久成员属性, 有一个 private 实例变量 firstName, 这个类定义了一个 getFirstName 和 setFirstName 方法检索和设置 firstName 实例变量的状态。

单值的持久成员属性的方法签名如下:

Type getProperty()

Void setProperty(Type type)

Collection-valued 持久性数据成员和成员属性必须使用支持 Java 的集合接口, 不论实体使用持久性数据成员或成员属性。以下是可用的集合接口:

(1) java.util.Collection

(2) java.util.Set

(3) java.util.List

(4) java.util.Map

如果实体类使用持久性数据成员, 上述方法签名必须是以上集合类型中的某一个。这些集合类型的 Generic variants 也可使用。例如, 如果实体 Customer 有一个持久性成员属性, 其中包含了一串电话号码, 可以使用以下方法:

Set <PhoneNumber> getPhoneNumbers(){ }

Void setPhoneNumbers(Set<PhoneNumber>){ }

在对象/关系中映射的注解必须应用于 getter 方法。映射注解不能应用于注解为 @Transient 或标记为 transient 的数据成员或成员属性。

3. 实体中的主键

每个实体都具有唯一的对象标识符。例如 customer 实体, 可能用 customer number 标识。实体的唯一标识符或主键, 确定唯一的实体实例。每一个实体必须有一个主键。一个实

体可以有简单或复合主键。

简单的主键使用 `javax.persistence.Id` 注解来表示主键属性或字段。

复合主键必须符合任何一个单一的持久属性或字段，或一组单一的持久性属性或字段。复合主键必须定义在一个主键类中。复合主键用 `javax.persistence.EmbeddedId` 和 `javax.persistence.IdClass` 注解表示。

主键或复合主键的字段或属性都必须是以下的 Java 语言类型：

- (1) Java primitive types。
- (2) Java primitive wrapper types。
- (3) `java.lang.String`。
- (4) `java.util.Date`（时间类型应为 `DATE`）。
- (5) `java.sql.Date`。

主键不能是浮点类型。如果使用生成主键，只有积分类型被携带。

主键类必须满足以下要求：

- (1) 这个类的访问控制修饰符必须是 `public`。
- (2) 如果基本属性访问可用，主键类的成员属性必须是 `public` 或 `protected`。
- (3) 这个类必须有一个 `public` 的默认构造方法。
- (4) 这个类必须实现 `hashCode()`方法和 `equals (Object other)`方法。
- (5) 这个类必须是可序列化的。
- (6) 复合主键必须是代表和映射到实体类的多个数据成员或属性，或必须代表和映射为一个嵌入类。
- (7) 如果这个类映射到实体类的多个数据成员或属性，主键类的主键数据成员或属性的名称和类型必须与实体类匹配。

下面的主键类是复合键，数据成员 `orderId` 和 `itemId` 一起唯一识别一个实体。

【例 7-1】主键类。

```
public final class LineItemKey implements Serializable{
    public Integer orderId;
    public int itemId;
    public LineItemKey(){}
    public LineItemKey(Integer orderId,int itemId){
        this.orderId=orderId;
        this.itemId=itemId;
    }
    public boolean equals(Object otherObj){
        if(this==otherObj){
            return true;
        }
        if(!(otherObj instanceof LineItemKey)){
            return false;
        }
        LineItemKey other=(LineItemKey) otherObj;
        return(
```

```

        (orderId==null?other.orderId==null:orderId.equals
        (other.orderId)
    )
    &&
    (itemId==other.itemId)
);
}
public int hashCode(){
    return(
        (orderId==null?0:orderId.hashCode())
        ^
        ((int)itemId)
    );
}

public String toString(){
    return""+orderId+"-"+itemId;
}
}

```

4. 多重的实体关系

有四种类型的实体关系：一对一，一对多，多对一，以及多对多。

(1) 一对一：每个实体与另一个单独的实体有关。例如，建立一个每个 `StorageBin` 中包含一个 `widget` 的物理仓库，`StorageBin` 和 `widget` 就是一个一对一的关系。一对一关系使用 `javax.persistence.OneToOne` 注解相应的持久属性或成员（property or field）。

(2) 一对多：一个实体实例能够被另一个实体的多个实例关联。例如，一个 `sales order`，可以有多个 `lineitems`。在该 `order` 的应用中，`Order` 和 `LineItem` 就是一个一对多的关系。一对多关系使用 `javax.persistence.OneToMany` 注解相应的持久属性或成员（property or field）。

(3) 多对一：一个实体的多个实例和另一实体的单个实例关联。这和一对多关系相反。在刚才提到的例子中，`LineItem` 与 `Order` 的关系是多对一。多对一关系使用 `javax.persistence.ManyToOne` 注解相应的持久属性或成员（property or field）。

(4) 多对多：一个实体的多个实例与另一个实体的多个实例关联。例如，学院里每门课程有很多学生，每个学生选修若干课程。因此，在入学申请中，课程和学生是一个多对多的关系。多对多关系使用 `javax.persistence.ManyToMany` 注解相应的持久属性或成员。

5. 实体关系的方向

关系的方向可以是单向或双向的。一个双向的关系既有自身端又有对方端，单向关系只有自身端。关系的自身端决定在数据库中如何持续运行更新的关系。

(1) 双向关系。在双向的关系中，每个实体有一个关系成员或属性（field or property）指向另一个实体。通过关系成员或属性，一个实体类的代码可以访问相关联的对象。如果一个实体有一个关联的成员，这个实体就知道了它的相关对象。例如，如果 `Order` 知道了它的 `LineItem` 的实例，并且 `LineItem` 知道自己属于哪个 `Order`，那么它们就是一个双向的关系。

双向关系必须遵循下列规则：

- ① 双向关系的对方端必须使用 `mappedBy` 元素的 `@OneToOne`，`@OneToMany` 或 `@ManyToMany` 注解。`mappedBy` 元素指定的实体中的属性或成员就是关系的拥有者。
- ② 多对一双向关系的多方不能定义 `mappedBy` 元素。多方通常是关系的自身端。
- ③ 对于一对一的双向关系，自身端对应包含了相应的外键的一方。
- ④ 对于多对多的双向关系，任何一方都可以是自身端。

(2) 单向关系。在单向关系中，只有一个实体有一个关系成员或属性涉及另一个实体。例如，`LineItem` 有一个关系成员标识为 `Product`，但是 `Product` 没有关系成员或属性关联到 `LineItem`。换言之，`LineItem` 知道 `Product`，但 `Product` 不知道哪一个 `LineItem` 实例指向它。

(3) 查询和关系的方向。Java 持久性查询语言的查询常常跨越关系浏览实体。关系的方向决定查询是否可以从一个实体浏览另一个实体。例如，查询可以由 `LineItem` 浏览 `Product`，但不能从相反的方向浏览。对 `Order` 和 `LineItem` 而言，查询可在两个方向浏览，这是因为这两个实体是双向关系。

(4) 级联删除和关系。使用关系的实体往往与在关系中存在的其他实体有依赖关系。例如，`Lineitem` 是 `Order` 的一部分，如果该 `Order` 被删除，那么该 `Lineitem` 也应被删除。这就是所谓的级联删除关系。

级联删除关系是通过通过对 `@OneToOne` 和 `@OneToMany` 关系设置 `cascade=REMOVE` 节点来设定的。例如：

```
@OneToMany(cascade=REMOVE,mappedBy="customer")
public Set<Order> getOrders(){return orders;}
```

6. 实体的继承

实体支持类继承，多态关联和多态查询。它们可以扩展非实体类，非实体类也可以扩展实体类。实体类可以是抽象的也可以是具体的。

(1) 抽象实体。一个抽象类可以通过被 `@Entity` 注解修饰而被声明为一个实体，抽象实体和具体实体的唯一不同在于抽象实体不能被实例化。

抽象实体能够像具体实体一样被查询，如果抽象实体是一个查询的目标，查询会在抽象实体的所有具体子类上执行。

【例 7-2】抽象实体。

```
@Entity
Public abstract class Employee{
    @Id
    Protected Integer employeeId;
    ...
}
@Entity
Public class FullTimeEmployee extends Employee{
    Protected Integer salary;
    ...
}
@Entity
Public class PartTimeEmployee extends Employee{
```

```

        Protected Float hourlyWage;
    }

```

(2) 被映射的父类。实体可能是继承自包含持久性状态和映射信息的父类，但父类不是实体，这是因为父类没有用 `@Entity` 注解修饰，并且也没有被 Java 持久性提供者映射为实体。这种情况通常出现在当要用父类来包含对多个实体子类共同的状态和映射信息的时候。

被映射的父类是通过 `javax.persistence.MappedSuperclass` 注解来定义的。

【例 7-3】 被映射的父类。

```

@MappedSuperclass
public class Employee{
    @Id
    protected Integer employeeId;
    ...
}
@Entity
public class FullTimeEmployee extends Employee{
    protected Integer salary;
    ...
}
@Entity
public class PartTimeEmployee extends Employee{
    protected Float hourlyWage;
    ...
}

```

被映射的父类是不可查询的，不能用于 `EntityManager` 或 `Query` 的操作，必须在 `EntityManager` 或 `Query` 的操作中使用被映射的父类的实体子类。被映射的父类不能是实体关系的目标，被映射的父类可以是抽象的或具体的。

被映射的父类没有对应的表格，继承被映射的父类的实体子类定义表格映射。例如，上述代码中底层表应该是 `FullTimeEmployee` 和 `PartTimeEmployee`，而 `Employee` 表是不存在的。

(3) 非实体的父类。实体可以有非实体的父类，并且这些父类可以是抽象的或具体的。非实体父类的状态是非持久性的，并且任何继承自非实体父类的状态也是非持久性的。非实体父类不能用于 `EntityManager` 或 `Query` 的操作，非实体父类中的任何映射或关系注解都将被忽略。

(4) 实体继承映射策略。通过对根类设置 `javax.persistence.Inheritance` 注解来定义 Java 持久性提供者如何将继承实体映射到底层数据库。有三种映射策略被用来映射实体数据到底层数据库：

- ① 一个单独的表对应类层次结构。
- ② 一个表对应具体实体类。
- ③ “联合”策略，子类的与父类不同的域或属性被映射到与父类不同的表。

【例 7-4】 实体继承映射策略。

```

public enum InheritanceType{

```

```
SINGLE_TABLE,
JOINED,
TABLE_PER_CLASS
};
```

默认映射策略是 `InheritanceType.SINGLE_TABLE`，在实体层次结构中的根类未设置 `@Inheritance` 注解的情况下使用。

第一种一个单独的表对应类层次结构策略：使用这种策略时，对应的 `javax.persistence.InheritanceType` 枚举类型值是 `InheritanceType.SINGLE_TABLE`，所有处于层次结构中的类都被映射到数据库中一个单独的表，这个表有一个列用做识别符，这个列存储不同的值来区分不同的行代表的不同实体的不同子类。

识别符列可以在实体类层次结构的根用 `javax.persistence.DiscriminatorColumn` 注解来定义。

`javax.persistence.DiscriminatorType` 枚举类型是用来设定识别符列的类型，可以通过设置 `@DiscriminatorColumn` 注解的 `discriminatorType` 节点值来实现，详见表 7-1。

表 7-1 @DiscriminatorColumn 节点

类 型	名 称	描 述
String	Name	识别符列的名称，默认是 <code>DTYPE</code> ，此节点是可选的
DiscriminatorType	discriminatorType	识别符列的类型，默认是 <code>DiscriminatorType.STRING</code> ，此节点是可选的
String	columnDefinition	创建识别符列的 SQL 片断，默认是由持久性提供者生成且是执行相关的，此节点是可选的
String	Length	基于字符串类型的识别符列的列宽度，对于不是字符串类型的识别符列，此节点被忽略。此节点是可选的

【例 7-5】DiscriminatorType 的定义。

```
public enum DiscriminatorType{
    STRING,
    CHAR,
    INTEGER
};
```

如果 `@DiscriminatorColumn` 在实体结构层次的根没有设定，需要创建识别符列时，持久性提供者会自动创建一个名为 `DTYPE` 的识别符列，类型默认为 `DiscriminatorType.STRING`。

`javax.persistence.DiscriminatorValue` 注解用于设置类层次结构中不同类在识别符列中的值，可以只用 `@DiscriminatorValue` 注解修饰具体实体类。

如果需要创建识别符列的实体未设定 `@DiscriminatorValue`，持久性提供者会提供一个默认的、实现相关的值。如果 `@DiscriminatorColumn` 的 `discriminatorType` 节点的值是 `DiscriminatorType.STRING`，则默认的值是实体的名称。

这种策略为整个实体类层次结构的实体和查询之间的多态关系提供了良好的支持，然而，它要求包含子类状态的列不能为 `null` 值。

第二种：一个表对应的具体实体类：对应于 `InheritanceType.TABLE_PER_CLASS`，每个具体类映射到数据库中的一个单独的表。类的所有域和属性（包含继承的域和属性）映射到表的各列。

这种策略对多态关系的支持性很差，对于针对整个类层次结构的查询通常要求使用 `SQLUNION` 查询或对每个子类单独查询。

这种策略是可选的，并且不被所有的 Java 持久性提供者支持，默认的应用服务器端持久性提供者不支持这种策略。

第三种：联合子类策略：对应于 `InheritanceType.JOINED`，根类由一个单独的表映射，每个子类有一个单独的表，仅包含子类自己独有的域。也就是说，子类表不包含继承的属性或值，子类表也有一列或数列代表自己的主键，它同时也是父类主键的外键。

这种策略对多态关系的支持良好，但当初始化实体子类时要求一个或多个联合操作，这有可能会因为密集类层次结构而导致很差的工作表现。类似地，覆盖整个类层次结构的查询由于要求子类表之间的联合操作，也会导致工作表现下降。

有的 Java 持久性提供者，包含应用服务器端的默认提供者，要求在使用联合子类策略时数据库表中必须有一个识别符列来对应根实体。如果程序没有使用自动表格创建，那么应确保数据库表的识别符列创建正确，或者可以使用 `@DiscriminatorColumn` 注解来配合数据库表结构。

7.1.2 管理实体

实体是由实体管理器来管理的，实体管理器由 `javax.persistence.EntityManager` 实例来代表，每一个实体管理器实例均与一个持久性上下文相关，一个持久性上下文定义了一个特定的实体实例的范围。

一个持久性上下文是一组存在于特定数据存储里的受管理的实体实例。`EntityManager` 接口定义了用于与持久性上下文交互的方法。

1. EntityManager接口

`EntityManager` API 创建和删除持久性实体实例，通过实体主键来查找实体，允许在实体上运行查询。

(1) 受容器管理的实体管理器。对于一个受容器管理的（Container-managed）实体管理器，一个 `EntityManager` 实例的持久性上下文是自动由容器向所有在一个单独的 JTA 事务内使用 `EntityManager` 实体的应用程序组件广播的。

JTA 事务通常包含了跨越多个应用程序组件的调用。要完成一个 JTA 事务，这些组件通常需要取得一个单独的持久性上下文。这发生在当一个 `EntityManager` 通过使用 `javax.persistence.PersistenceContext` 注解而被注入到应用程序组件中时。持久性上下文自动在当前 JTA 事务内被广播，并且被映射到同一持久性单元的 `EntityManager` 引用提供入口进入当前事务的持久性上下文。通过自动广播持久性上下文，应用程序组件就不需要在一个单独的 JTA 事务里互相传递 `EntityManager` 实例的引用。JavaEE 容器负责管理受容器管理的实体管理器的生命周期。

要获得一个 `EntityManager` 实例，需要将实体管理器注入到应用程序组件中。

```
@PersistenceContext
EntityManager em;
```

(2) 受应用程序管理的实体管理器。对于一个受应用程序管理的（Application-managed）实体管理器，恰恰相反，持久性上下文是不对应用程序组件广播的，`EntityManager` 实例的生

命周期是由应用程序管理的。

在这种情况下，每一个 `EntityManager` 创建一个新的、独立的持久性上下文。`EntityManager`，以及与之关联的持久性上下文，是由应用程序分别创建和销毁的。

这种情况下，应用程序通过使用 `javax.persistence.EntityManagerFactory` 的 `createEntityManager` 方法来创建 `EntityManager` 实例。

要获得一个 `EntityManager` 实例，首先必须通过 `javax.persistence.PersistenceUnit` 注解将一个 `EntityManagerFactory` 实例注入应用程序组件：

```
@PersistenceUnit
EntityManagerFactory emf;
```

然后，从一个 `EntityManagerFactory` 实例获得 `EntityManager`：

```
EntityManager em=emf.createEntityManager();
```

(3) 通过 `EntityManager` 查找实体。`EntityManager.find` 方法用于通过主键查找实体。

【例 7-6】受应用程序管理的实体管理器。

```
@PersistenceContext
EntityManagerem;

Public void enterOrder(int custID,Order newOrder){
    Customer cust=em.find(Customer.class,custID);
    cust.getOrders().add(newOrder);
    newOrder.setCustomer(cust);
}
```

(4) 管理实体实例的生命周期。通过 `EntityManager` 实例来调用实体操作从而管理实体，实体实例有四种状态：新创建的、受管理的、游离的和删除的。

新创建的（New）实体实例：没有持久性标识，也没有与之关联的持久性上下文。

受管理的（Managed）实体实例：有持久性标识，也有与之关联的持久性上下文。

游离的（Detached）实体实例：有持久性标识，但目前没有与之关联的持久性上下文。

删除的（Removed）实体实例：有持久性标识，也有与之关联的持久性上下文，但计划将其从数据存储中删除。

① 持久性实体实例。新创建的实体实例转变到受管理状态和持久性可以通过两种方法实现：通过调用 `persist` 方法，或者通过有关联的、在关系注解中设置了 `cascade=PERSIST` 或 `cascade=ALL` 的实体调用级联的 `persist` 操作。这意味着当与 `persist` 操作的相关事务完成时实体的数据已存入数据库。如果实体已经是受管理状态，`persist` 操作将被忽略，然而 `persist` 操作会级联到有关联的、在关系注解中设置了 `cascade=PERSIST` 或 `cascade=ALL` 的实体。如果 `persist` 是在一个被删除的实体上操作的，则该实体会转换到受管理状态。如果实体是游离的，`persist` 方法会引发一个 `IllegalArgumentException` 异常，或造成事务提交失败。

【例 7-7】持久性实体实例。

```
@PersistenceContext
EntityManager em;
...

Public LineItem createLineItem(Orderorder,Productproduct,intquantity){
    LineItem li=new LineItem(order,product,quantity);
```

```

        order.getLineItems().add(li);
        em.persist(li);
        return li;
    }

```

Persist 操作会向有关联的、在关系注解中设置了 `cascade=PERSIST` 或 `cascade=ALL` 的实体广播。

【例 7-8】Persist 操作向实体广播。

```

@OneToMany(cascade=ALL,mappedBy="order")
Public Collection<LineItem> getLineItems(){
    Return lineItems;
}

```

② 删除实体实例。删除受管理的实体实例可以通过两种方法实现：调用 `remove` 方法；通过有关联的、在关系注解中设置了 `cascade=REMOVE` 或 `cascade=ALL` 的实体调用级联的 `remove` 操作。如果 `remove` 操作是在一个新创建的实体上调用的，`remove` 操作将被忽略，然而 `remove` 操作会级联到有关联的、在关系注解中设置了 `cascade=REMOVE` 或 `cascade=ALL` 的实体。如果 `remove` 操作是在游离的实体上调用的，`remove` 方法会引发一个 `IllegalArgumentException` 异常，或造成事务提交失败。如果 `remove` 操作是在已被删除的实体上调用的，它将被忽略。当事务完成时，或当执行完 `flush` 操作后，实体的数据将从数据存储中删除掉。

【例 7-9】删除实体实例。

```

Public void removeOrder(IntegerorderId){
    try{
        Orderorder=em.find(Order.class,orderId);
        em.remove(order);
    }...
}

```

在这个例子中，所有与 `order` 有关的 `LineItem` 实体都将被删除，因为 `Order.getLineItems` 中在关系注解中设置了 `cascade=ALL` 注解。

③ 将实体数据与数据库同步。持久性的实体的状态在与之相关的事务提交时实现与数据库同步。如果一个受管理的实体与另一个受管理的实体是双向关系，数据将会基于此关系而保持持久性。要强迫实体数据与数据库同步，调用实体的 `flush` 方法。如果该实体与另外的实体有关系，并且关系注解中将 `cascade` 节点设为 `PERSIST` 或 `ALL`，调用 `flush` 时关联的实体数据也会与数据库同步。如果实体被删除了，调用 `flush` 将会从数据库中删除该实体的所有数据。

(5) 创建查询。`EntityManager.createQuery` 和 `EntityManager.createNamedQuery` 方法是用来为 Java 持久性查询语言创建查询的。

`createQuery` 方法用于创建动态查询，直接定义于应用程序业务逻辑范围的查询。

【例 7-10】createQuery 方法创建动态查询。

```

public List findWithName(Stringname){
    returnem.createQuery(
        "SELECTcFROMCustomercWHEREc.nameLIKE:custName")
        .setParameter("custName",name)
        .setMaxResults(10)

```



```

        .getResultList();
    }

```

`createNamedQuery` 方法用于创建静态查询，用 `javax.persistence.NamedQuery` 注解在元数据里定义的查询。`@NamedQuery` 注解的 `name` 节点定义了 `createNamedQuery` 方法将会使用的查询的名字。`@NamedQuery` 注解的 `query` 节点就是查询本身。

【例 7-11】 `@NamedQuery` 注解的 `query` 节点。

```

@NamedQuery(
    name="findAllCustomersWithName",
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName"
)

```

下面是 `createNamedQuery` 方法的例子，使用了上面的 `@NamedQuery` 注解。

【例 7-12】 使用了 `@NamedQuery` 注解的 `createNamedQuery` 方法。

```

@PersistenceContext
public EntityManager em;
...

customers=em.createNamedQuery("findAllCustomersWithName")
    .setParameter("custName","Smith")
    .getResultList();

```

① 查询中的命名 (Named) 参数。命名参数是查询中前缀有 “:” 号的参数。查询中的命名参数是与 `javax.persistence.Query.setParameter(Stringname, Objectvalue)` 方法中的一个参数相绑定的。在下面的例子中，`findWithName` 业务方法的 `name` 参数在查询中通过调用 `Query.setParameter` 方法时与 `custName` 命名参数相绑定的。

【例 7-13】 查询中的 Named 参数。

```

Public List findWithName(String name){
    return em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE :custName")
        .setParameter("custName",name)
        .getResultList();
}

```

命名参数是大小写敏感的，可以用在动态或静态查询中。

② 查询中的占位 (Positional) 参数。在查询中可以交替使用占位参数，而非命名参数。占位参数是前缀一个 “?” 号，然后是它在查询中的位置序号。`Query.setParameter(integerposition, Objectvalue)` 方法就是用来设定这种参数的值的。

在下面的例子中，`findWithName` 业务方法被用输入参数重写了。

【例 7-14】 被重写的 `findWithName` 业务方法。

```

Public List findWithName(Stringname){
    return em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE ?1")
        .setParameter(1,name)
        .getResultList();
}

```

输入参数从 1 开始计数，大小写敏感，可以用在动态或静态查询中。

2. 持久性单元

一个持久性单元定义了一组在应用程序中受 `EntityManager` 实例管理的持久性类。这组实体类代表了存储在数据库中的数据。

持久性单元是通过 `persistence.xml` 设置文件来定义的。包含 `persistence.xml` 文件的 JAR 文件或 `META-INF` 目录被称为持久性单元的根，持久性单元的范围决定于持久性单元的根。

每个持久性单元都必须用一个在持久性单元范围内独一无二的名称标识。

持久性单元可以作为一个 WAR 或 EJBJAR 文件的一部分来打包，或先打包成一个 JAR 文件，然后再包含到 WAR 或 EAR 文件中。

如果将持久性单元作为一组类打包到一个 EJBJAR 文件中，`persistence.xml` 文件应该放到 EJBJAR 文件的 `META-INF` 目录中。

如果将持久性单元作为一组类打包到一个 WAR 文件中，`persistence.xml` 文件应该放到 WAR 文件的 `WEB-INF/classes/META-INF` 目录中。

如果将持久性单元作为一个将要被包括到 WAR 或 EAR 文件中的 JAR 文件来打包，JAR 文件应该放在以下位置：WAR 文件的 `WEB-INF/lib` 目录，EAR 文件的顶端，EAR 文件的库目录。

`persistence.xml` 文件定义一个或多个持久性单元。

【例 7-15】 `persistence.xml` 文件。

```
<persistence>
  <persistence-unit name="OrderManagement">
    <description>This unit manages orders and customers.
      It does not rely on any vendor-specific features and can
      Therefore be deployed to any persistence provider.
    </description>
    <jta-data-source>jdbc/MyOrderDB</jta-data-source>
    <jar-file>MyOrderApp.jar</jar-file>
    <class>com.widgets.Order</class>
    <class>com.widgets.Customer</class>
  </persistence-unit>
</persistence>
```

这个文件定义了一个叫 `OrderManagement` 的持久性单元，使用了一个 JTA 感知的数据源 `jdbc/MyOrderDB`。`jar-file` 和 `class` 节点定义了受管理的持久性类：实体类、嵌入类和被映射的父类。`jar-file` 节点定义了对包含了受管理的持久性类的、打包了的持久性单元透明的 JAR 文件，同时，`class` 节点则分别为受管理的持久性类命名。

`jta-data-source`（针对 JTA 感知的数据源）和 `non-jta-data-source`（针对非 JTA 感知的数据源）节点定义了容器使用的数据源的全局性 JNDI 名称。

7.2 Web层持久性

本部分介绍如何在 Web 应用程序中使用 Java 持久性 API。重点是源代码和一个例子“书店”的相关设置。例子“书店”是一个 Web 应用程序，管理与书店相关的实体。

在 Web 组件之间共享和在一个 Web 应用程序各个调用之间共享的数据通常保存在一个

数据库中。Web 应用程序通过 Java 持久性 API（见 7.1）来访问关系数据库。

Java 持久性 API 提供了一组管理 Java 对象与持久性数据（存储在数据库中）之间对象/关系映射（ORM）的工具。一个映射到数据库表的 Java 对象被称为实体类。这是一个标准的 Java 对象（也称为 POJO，或简单 Java 对象），具有映射到数据库表列的属性。“杜克大学书店”应用程序有一个实体类 `Book`，映射到 `WEB_BOOKSTORE_BOOKS`。

Java 持久性工具管理实体之间的交互，应用程序是通过使用 `EntityManager` 接口来实现的。这个接口提供方法用于执行一般性的数据库功能，如查询和更新数据库。“杜克大学书店”应用程序中的 `BookDBAO` 类使用 `EntityManager` 来查询数据库的图书资料和更新已出售书籍清单。

这套可以用实体管理器管理的实体是在一个持续性单元里定义的。它监督应用程序中所有的持久性操作。持久性单元是由一个叫 `persistence.xml` 的文件所定义的，该文件还规定了数据源、事务类型以及其他信息。在“杜克大学书店”应用程序中，`persistence.xml` 文件和 `Book` 类被打包到一个单独的 JAR 文件中，并将其添加到应用程序的 WAR 文件中。

和 JDBC 技术一样，一个数据源对象有一组属性，用以确定和描述它所代表的现实世界的数据库，这些属性包括数据库服务器的位置、数据库名称，用于与服务器通信的网络协议等信息。

一个应用程序使用 Java 持久性 API 并不需要明确建立一个到数据源的连接，而在 JDBC 技术中则要求如此。尽管如此，必须在应用服务器端建立 `DataSource` 对象。

7.2.1 定义持久性单元

`persistence.xml` 文件包装在应用程序的 WAR 文件中。这个文件包括以下内容：

（1）一个持久性节点，确定文件所验证的架构和包含持久性单元的节点。一个持久性单元节点确认持久性节点的名称和事务类型。

（2）一个可选的说明性节点。

（3）一个 `jta-data-source` 节点，负责定义全局性的 JNDI 名称和 JTA 数据源。

`jta-data-source` 节点指明实体管理器参与的事务是 JTA 的事务，这意味着事务是由容器管理的。或者，可以使用 `resource-local` 事务，这类事务由应用程序本身控制。一般情况下，Web 应用程序开发人员使用 JTA 的事务，这样，可以不需要手动管理 `EntityManager` 实例的生命周期。一个 `resource-local` 管理器不能参与全局性事务。此外，Web 容器将不会回滚因为代码写得很差而不能完成的事务。

7.2.2 创建一个实体类

一个实体类就是一个代表数据库中一个表的组件。在“杜克大学书店”这个例子中，只有一个数据库表，因此只有一个实体类：`Book`。

`Book` 类包含可以获取特定书籍各部分数据的属性，如这本书的标题和作者。要使它成为一个实体管理器可以管理的实体类，需要做到以下几点：

（1）为类（本身）添加 `@Entity` 注解。

（2）为映射表主键的属性添加 `@Id` 注解。

（3）如果数据库表名称和实体类名不同的话，为类添加 `@Table` 注解，以便识别数据库表名称。

(4) 有选择地使用类序列化。

【例 7-16】 Book 类的部分代码。

```
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name="WEB_BOOKSTORE_BOOKS")
public class Book implements Serializable {
    private String bookId;
    private String title;
    public Book() { }
    public Book(String bookId, String title, ...) {
        this.bookId = bookId;
        this.title = title;
        ...
    }
    @Id
    public String getBookId() {
        return this.bookId;
    }
    public String getTitle() {
        return this.title;
    }
    ...
    public void setBookId(String id) {
        this.bookId=id;
    }
    public void setTitle(String title) {
        this.title=title;
    }
    ...
}
```

7.2.3 获取对一个实体管理器的访问

“杜克大学书店”应用程序中的 BookDBAO 对象包含了从数据库中得到书本有关数据和更新图书已出售时数据库中的书本库存的方法。为了执行数据库查询，BookDBAO 对象需要获得一个 EntityManager 实例。

Java 持久性 API 允许开发者使用注解以确定一个源，以便容器可以透明地把它注入到一个对象中，可以通过使用 @PersistenceUnit 注解注入一个 EntityManagerFactory，从而使一个对象获得一个 EntityManager 实例。

对于 Web 应用程序开发者来说，不便的是，通过注解进行源注入只适用于可以用兼容 Java EE 标准容器管理的类。由于 Web 容器不管理 JavaBeans 组件，不能将源注入它们。唯一的例外是请求范围内的 JavaServerFaces 管理的 Bean，它们是由容器管理的，因此，支持

源注入。这仅仅在应用程序是一个 `JavaServerFaces` 应用时有用。

如果能在一个由容器管理的对象里做到源注入的话，仍然可以在一个不是 `JavaServerFaces` 的 `Web` 应用程序中使用源注入。这些对象包括 `Servlets` 和 `ServletContextListener` 对象。这些对象可以使应用程序的 `Bean` 利用这些源。

在“杜克大学书店”这个例子中，该 `ContextListener` 对象创建 `BookDBAO` 对象，使之适用于应用程序的整个范围。在这个过程中，它向 `BookDBAO` 对象传递已经注入了 `ContextListener` 的 `EntityManagerFactory` 对象。

【例 7-17】 获取对一个实体管理器的访问。

```
public final class ContextListener implements ServletContextListener {
...

@PersistenceUnit
private EntityManagerFactory emf;

public void contextInitialized(ServletContextEvent event) {
    context = event.getServletContext();
    ...
    try {
        BookDBAO bookDB = new BookDBAO(emf);
        context.setAttribute("bookDB", bookDB);
    } catch (Exception ex) {
        System.out.println(
            "Couldn't create bookstore database bean: "
                + ex.getMessage());
    }
}
}
```

`BookDBAO` 对象能够从 `ContextListener` 对象传递过来的 `EntityManagerFactory` 那里得到一个 `EntityManager` 对象：

```
private EntityManager em;
public BookDBAO (EntityManagerFactory emf) throws Exception {
    em = emf.getEntityManager();
    ...
}
```

不过 `JavaServerFaces` 版本的“杜克大学书店”例程在获取 `EntityManager` 实例的方式上有所不同，因为受管理的 `Bean` 可以进行源注入，所以可以直接将 `EntityManagerFactory` 实例注入到 `BookDBAO` 对象中。

事实上，可以略过注入 `EntityManagerFactory` 实例，而将 `EntityManager` 直接注入 `BookDBAO` 对象，这是因为线程安全与请求范围内的 `Bean` 无关。与之相反的是，在涉及到 `servlets` 和 `listeners` 时开发者应该关注线程安全，因此，一个 `servlets` 或 `listeners` 需要注入一个 `EntityManagerFactory` 实例（`EntityManagerFactory` 本身是线程安全的），而 `persistencecontext` 不是线程安全的。

下面是 `JavaServerFaces` 版本的“杜克大学书店”例程中 `BookDBAO` 对象的部分代码。

【例 7-18】 `BookDBAO` 对象的部分代码。

```
import javax.ejb.*;
import javax.persistence.*;
import javax.transaction.NotSupportedException;

public class BookDBAO {
    @PersistenceContext
    private EntityManager em;
```

正如前面的代码所显示的一样，EntityManager 实例是被注入到一个添加了 @PersistenceContext 注解的对象当中，EntityManager 实例是和受 EntityManager 管理的一组实体实例组成的 persistencecontext 相关联的。

注解可以确认与之相关联的持久性单元的名称，这个名称必须和 persistence.xml 文件中定义的持久性单元相吻合。

7.2.4 访问数据库中的数据

当 BookDBAO 对象获取一个 EntityManager 实例后，它就可以从数据库中获得数据了。BookDBAO 对象的 getBooks 方法调用 EntityManager 实例的 createQuery 方法来按照 bookId 查询书本清单。

【例 7-19】访问数据库中的数据。

```
public List getBooks() throws BooksNotFoundException {
    try {
        return em.createQuery(
            "SELECT bd FROM Book bd ORDER BY bd.bookId").
            getResultList();
    } catch (Exception ex) {
        throw new BooksNotFoundException("Could not get books: "
            + ex.getMessage());
    }
}
```

BookDBAO 对象的 getBook 方法调用 EntityManager 实例的 find 方法在数据库中查找特定的书本并返回相应的 book 对象的实例。

【例 7-20】在数据库中查找特定的书本并返回相应的 book 对象。

```
public Book getBook(String bookId) throws BookNotFoundException {
    Book requestedBook = em.find(Book.class, bookId);
    if (requestedBook == null) {
        throw new BookNotFoundException("Couldn't find book: "
            + bookId);
    }
    return requestedBook;
}
```

7.2.5 更新数据库中的数据

在“杜克大学书店”程序中，更新数据库涉及当用户购买数本某书时递减一本书的存货数目，BookDBAO 对象通过调用 buyBooks 和 buyBook 方法执行此操作。

【例 7-21】更新数据库中的数据。

```

public void buyBooks(ShoppingCart cart) throws OrderException{
    Collection items = cart.getItems();
    Iterator i = items.iterator();
    try {
        while (i.hasNext()) {
            ShoppingCartItem sci = (ShoppingCartItem)i.next();
            Book bd = (Book)sci.getItem();
            String id = bd.getBookId();
            int quantity = sci.getQuantity();
            buyBook(id, quantity);
        }
    } catch (Exception ex) {
        throw new OrderException("Commit failed: "
            + ex.getMessage());
    }
}

public void buyBook(String bookId, int quantity)
    throws OrderException {
    try {
        Book requestedBook = em.find(Book.class, bookId);
        if (requestedBook != null) {
            int inventory = requestedBook.getInventory();
            if ((inventory - quantity) >= 0) {
                int newInventory = inventory - quantity;
                requestedBook.setInventory(newInventory);
            } else{
                throw new OrderException("Not enough of "
                    + bookId + " in stock to complete order.");
            }
        }
    } catch (Exception ex) {
        throw new OrderException("Couldn't purchase book: "
            + bookId + ex.getMessage());
    }
}

```

在 `buyBook` 方法中，通过调用 `EntityManager` 实例的 `find` 方法来取得购物车中的某本书，然后更新 `book` 对象的存货数量。

为了确保更新被完整地提交和执行，`buyBook` 方法是在一个单独的事务里提交的。在 JSP 版本的“杜克大学书店”例程里，`Dispatcherservlet` 调用 `buyBook` 方法，因此设定交易划分。

在下面的代码中，`UserTransaction` 源被注入到 `Dispatcherservlet` 中。`UserTransaction` 是用于开始和结束一个事务的底层 JTA 事务管理器的一个接口。当得到 `UserTransaction` 源后，`Servlet` 调用 `UserTransaction` 的 `begin` 和 `commit` 方法来标明事务的边界，`UserTransaction` 的 `rollback` 方法则确保事务内的所有语句均被回滚以保证数据的完整性。

【例 7-22】 确保事务内的所有语句均被回滚以保证数据的完整性。

```
@Resource
UserTransaction utx;
...
try {
    utx.begin();
    bookDBAO.buyBooks(cart);
    utx.commit();
} catch (Exception ex) {
    try {
        utx.rollback();
    } catch (Exception exe) {
        System.out.println("Rollback failed: "+exe.getMessage());
    }
}
```

7.3 EJB层的持久性（多表实体Bean）

本节描述企业 Beans（多表实体 Bean）如何使用 Java 持久性 API，材料着重于例程的源码和设置。例程 roster 管理一个社区体育系统。本章假设已经对 6.1 节介绍的内容很熟悉。

Roster 应用程序维护休闲体育联盟中队员花名册。程序有 4 个组件：Java 持久性 API 实体（Player，Team，和 League）、一个有状态会话 Bean（RequestBean）、一个客户端应用程序（RosterClient）和三个帮助类（PlayerDetails，TeamDetails，and LeagueDetails）。

从功能上，roster 程序和 order 程序类似，但是有 3 个新特点：多对多关系，实体继承，程序部署时自动创建表格。

1. Roster程序中的关系

一个休闲运动体系有着下列关系：

- (1) 一个运动员，可以属于很多球队；
- (2) 一个球队，可以有很多球员；
- (3) 一个球队在某一个联盟里；
- (4) 一个联盟有很多球队。

在 roster 程序中上述关系都由 Player，Team 和 League 实体之间的关系得到反映。

- (1) Player 和 Team 之间是多对多的关系；
- (2) Team 和 League 之间是多对一的关系。

roster 程序中的多对多关系：

Player 和 Team 之间多对多的关系是通过使用 @ManyToMany 注解来确认的。

在 Team.java 文件中，@ManyToMany 注解修饰了 getPlayers 方法。

【例 7-23】 roster 程序中的多对多关系。

```
@ManyToMany
@JoinTable(
    name="EJB_ROSTER_TEAM_PLAYER",
```



```

joinColumns=
    @JoinColumn(name="TEAM_ID", referencedColumnName="ID"),
inverseJoinColumns=
    @JoinColumn(name="PLAYER_ID", referencedColumnName="ID")
)
public Collection<Player> getPlayers() {
    return players;
}

```

`@JoinTable` 注解是用来定义数据库中将 `playerID` 和 `teamID` 关联起来的表，定义 `@JoinTable` 注解的实体是关系的所有者，所以在这个例子中 `Team` 实体是和 `Player` 实体关系的所有者。因为 `Roster` 程序在部署时自动创建表，所以容器会在数据库中创建一个名叫 `EJB_ROSTER_TEAM_PLAYER` 的表。

`Player` 在和 `Team` 关系中是非所有方。与在一对一关系和多对一关系中一样，非所有方是用 `mappedBy` 节点来标明的。因为 `Palyer` 和 `Team` 之间的关系是双向的，选择谁做关系的所有者是可以随意的。

在 `Player.java` 文件中，`@ManyToMany` 注解修饰 `getTeams` 方法：

```

@ManyToMany(mappedBy="players")
public Collection<Team> getTeams() {
    return teams;
}

```

2. roster应用程序中的实体继承

`roster` 程序描述了如何使用实体继承。

`League` 实体是 `roster` 程序中的一个抽象类，有两个具体子类：`SummerLeague` 和 `WinterLeague`。因为 `League` 是一个抽象类，所以不能被实例化。

【例 7-24】 `roster` 应用程序中的实体继承。

```

...
@Entity
@Table(name = "EJB_ROSTER_LEAGUE")
public abstract class League implements java.io.Serializable {
    ...
}

```

与之对应的是，当用户要创建一个 `League` 对象的时候，他们会创建一个 `SummerLeague` 或 `WinterLeague` 对象。`SummerLeague` 和 `WinterLeague` 类继承了 `League` 类的持久性属性，只是增加了以 `sport` 为参数的构造方法，`sport` 参数验证是否与该 seasonal league 的 `sport` 参数类型相匹配。

【例 7-25】 `SummerLeague` 实体。

```

...
@Entity
public class SummerLeague extends League
    implements java.io.Serializable {

    /** Creates a new instance of SummerLeague */

```

```

public SummerLeague() {
}

public SummerLeague(String id, String name,
    String sport) throws IncorrectSportException {
    this.id = id;
    this.name = name;
    if (sport.equalsIgnoreCase("swimming") ||
        sport.equalsIgnoreCase("soccer") ||
        sport.equalsIgnoreCase("basketball") ||
        sport.equalsIgnoreCase("baseball")) {
        this.sport = sport;
    } else {
        throw new IncorrectSportException(
            "Sport is not a summer sport.");
    }
}
}

```

因为 roster 程序使用了默认的映射策略 `InheritanceType.SINGLE_TABLE`，所以 `@Inheritance` 注解就不必注明了。如果想使用不同的映射策略，用 `@Inheritance` 注解修饰 league，在 strategy 节点设定映射策略的值。

【例 7-26】 使用 `@Inheritance` 注解。

```

@Entity
@Inheritance(strategy=JOINED)
@Table(name="EJB_ROSTER_LEAGUE")
public abstract class League implements java.io.Serializable {
}

```

Roster 程序使用了默认的认识符列名称，所以 `@DiscriminatorColumn` 注解就不是必需的。因为现在使用的是自动表格创建，持久性提供者会在 `EJB_ROSTER_LEAGUE` 表中创建一个识别符列 `DTYPE`，用于存储创建 League 实体的继承实体的名称。如果想为识别符列取一个不同于默认的名称，用 `@DiscriminatorColumn` 注解修饰 League，在 name 节点中设定值。

【例 7-27】 使用 `@DiscriminatorColumn` 注解。

```

@Entity
@DiscriminatorColumn(name="DISCRIMINATOR")
@Table(name="EJB_ROSTER_LEAGUE")
public abstract class League implements java.io.Serializable{
...
}

```

3. roster程序里的自动表格生成

在部署时，应用服务器会自动删除和创建 roster 程序使用的数据库表，这是通过在 persistence.xml 文件里将 `toplink.ddl-generation` 属性设为 `drop-and-create-tables` 来实现的。

【例 7-28】 persistence.xml 文件。

```

<?xml version="1.0" encoding="UTF-8"?>

```

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
version="1.0">
    <persistence-unit name="em" transaction-type="JTA">
        <jta-data-source>jdbc/__default</jta-data-source>
        <properties>
            <property name="toplink.ddl-generation"
                value="drop-and-create-tables"/>
        </properties>
    </persistence-unit>
</persistence>
```

这个特性针对应用服务器使用的持久性提供者，在 Java EE 服务器间是不可移动的。表格自动创建在开发时是有用的，然而，当用于实际应用或迁移到其他 Java EE 服务器时，可以从 persistence.xml 文件中删除 toplink.ddl-generation 属性。

4. 编译和运行roster程序

要编译 order 程序的组件，输入下列命令：

```
ant
```

这个命令会执行默认的任务，编译源文件，并将程序打包到一个 tut-install/examples/ejb/roster/dist/roster.ear 文件中。

要布署这个 EAR 文件，保证应用服务器在运行，然后输入下列命令：

```
ant deploy
```

编译系统会检查 Java DB 数据库服务器是否在运行，如果没有则启动服务器，然后部署 roster.ear 文件。应用服务器会按 persistence.xml 文件中设定的那样删除或创建表格。

当 roster.ear 文件部署后，一个客户端文件 rosterClient.jar 生成了，它包含了程序的客户端。

要运行这个客户端，输入下列命令：

```
ant run
```

将看到以下输出：

```
[echo] running application client container.
[exec] List all players in team T2:
[exec] P6 Ian Carlyle goalkeeper 555.0
[exec] P7 Rebecca Struthers midfielder 777.0
[exec] P8 Anne Anderson forward 65.0
[exec] P9 Jan Wesley defender 100.0
[exec] P10 Terry Smithson midfielder 100.0

[exec] List all teams in league L1:
[exec] T1 Honey Bees Visalia
[exec] T2 Gophers Manteca
[exec] T5 Crows Orland
```

```
[exec] List all defenders:  
[exec] P2 Alice Smith defender 505.0  
[exec] P5 Barney Bold defender 100.0  
[exec] P9 Jan Wesley defender 100.0  
[exec] P22 Janice Walker defender 857.0  
[exec] P25 Frank Fletcher defender 399.0  
...
```

(1) all 任务：为方便起见，all 任务将会编译、打包、部署和运行程序，要做到这一点，输入下列命令：

```
ant all
```

(2) 卸载 roster：要卸载 roster.ear，输入下列命令：

```
ant undeploy
```

习 题 7

1. 实体类必须符合什么要求？
2. 什么是实体的主键？
3. 实体关系有哪些类型？实体关系的方向有哪些类型？各有什么特点？
4. 什么是持久性上下文，什么是持久性单元？
5. 简述如何在 Web 应用程序中使用 Java 持久性 API。
6. 简述如何在企业 Beans 使用 Java 持久性 API。

第 8 章 Web 服务与 SOA 技术

本章讨论 Web 服务 (Web Services) 的基本概念、如何用 JAX-WS (基于 XML 的 Web 服务的 Java API) 开发 Web 服务应用。通过在 Internet 或者在企业内部网或企业外部网上建立 Web 服务, 从而为应用提供相应的更多功能。开发标准 Web Services 的其他应用程序接口: JAXB (The Java API for XML Binding)、StAX (The Streaming API for XML) 以及 SAAJ (The SOAP with Attachments API for Java™), 本书不再详细讲解。

本章还将介绍面向服务结构 (SOA) 新技术: SOA 的概念、起源、理论、实现的路线以及未来的发展趋势。可以说, SOA 的出现, 将为整个企业级软件架构设计带来巨大的影响。

8.1 Web 服务到底是什么

一个 Web 服务到底是什么? 以及它将如何实现?

从表面上看, Web Service 就是一个应用程序, 它向外界暴露出一个能够通过 Web 进行调用的 API, 即用户能够用编程的方法通过 Web 调用来实现某个功能的应用程序。例如, 读者创建一个 Web Service, 它的作用是查询某学校某学生的基本信息。它将该学生的编号作为查询字符串, 返回该学生的具体信息。用户可以在浏览器的地址栏中直接输入 HTTP GET 请求来调用并在页面罗列该学生基本信息的 JSP, 这就可以算是体验 Web Service 了。

从深层次上来看, Web Service 是一种新的 Web 应用程序分支, 它们是自包含、自描述、模块化的应用, 可以在网络 (通常为 Web) 中被描述、发布、查找以及通过 Web 来调用。Web Service 便是基于网络的、分布式的模块化组件, 它执行特定的任务, 遵守具体的技术规范, 这些规范使得 Web Service 能与其他兼容的组件进行互操作。它可以使用标准的互联网协议, 如超文本传输协议 HTTP 和 XML, 将功能体现在互联网和企业内部网上。Web Service 平台是一套标准, 它定义了应用程序如何在 Web 上实现互操作性。

总之, 从本质上说, 一个 Web 服务即为一个远程组件, 其中包括了可以通过使用 Internet 协议来调用的方法, 这里最主要的 Internet 协议是 HTTP。传送的数据均为一个自述的 XML 文档进行保存。一个指定 Web 服务的功能可以用任何一种语言 (Java、C++、C、Visual Basic 等) 实现, 而且可以采用任何一种对象部署技术。唯一需要保证的是必须通过 HTTP 等 Internet 协议来访问。客户不用操心服务具体如何实现。

这样, 就可将一个 Web 服务认为是客户可以访问的功能, 它具体包括以下特点:

- (1) 语言无关性: 实现客户端的语言不必与实现服务的语言相一致。
- (2) 平台无关性: HTTP 相当于一个重要的平台均衡器。
- (3) 对象技术无关性: 客户不用知道, 也不用考虑对象是如何部署和管理的。

由于对 Web 服务的访问是通过 HTTP 等 Internet 协议实现的, 一个 Web 服务将自动具有以下优点:

(1) 防火墙和代理兼容性: 当前在 Internet 上实现应用集成时存在着一个问题, 即对于两个或多个公司之间的网络基础架构, 不允许在远程对象调用以及相应的应用集成中使用专用的协议和网络端口。具体来说, 如果存在防火墙和代理时就会出现这个问题, 因为许多防火墙

和代理都设计为需要确保只有端口 80 (HTTP) 数据流才能通过其防护层。基于 HTTP, 则可以使这些服务能够得以访问, 从而使它们有着同样的入口, 另外还不需要升级网络硬件。

(2) 自动 HTTP 鉴别: 由于这一特点是 HTTP 协议所带的, 因此所有基于 HTTP 的通信都可由此获益。

(3) 通过 SSL 实现加密通信: SSL 和 HTTP 的联合使用是保证数据在 Internet 上安全传输的一种有效的解决方案; Web 服务基于 Internet 协议部署时即可得到这一特性。

(4) HTTP1.1 持久连接: 对于基于 HTTP 的通信, 这是对性能的一种自动改进, 它是协议的一部分, 不需要 Web 服务提供者和使用者做任何工作。

作为一种组件技术, Web 服务还具有以下优点:

(1) 松耦合。客户不必与服务器或其组件技术紧密集成在一起。Web 服务建立在网络的 XML 型消息基础之上。

(2) 以编程方式访问。目前, 可以使用 Web 浏览器来执行远程方法, 如检查天气温度、股票价格和贷款利率等。不过通常需要以可视化方式来访问这些消息。通过编写代码的方式来访问此功能确实很困难, 特别是所返回的数据 (即一个 HTML Web 页面) 必须得到可靠的解析, 而内嵌的数据需要手工区别类型。但是智能代理和其他计算机程序则希望有一种更简单的办法可以通过程序来访问远程功能, 而 Web 服务正是雪中送炭。

以上所列出的所有特性在图 8-1 中都做了总结, 这里显示了一个生成报价的远程方法可以被远程访问。更具体地, 无论手动客户还是自动客户都可以通过诸如 HTTP 等 Internet 协议来调用远程对象功能, 并发出请求和接收响应。只需要知道其所需功能在哪里, 输入和输出的通信可以用一种与平台无关的方式——简单对象访问协议 (SOAP) 实现, 而内容则使用一种自描述的标记语言 (XML) 进行编码。

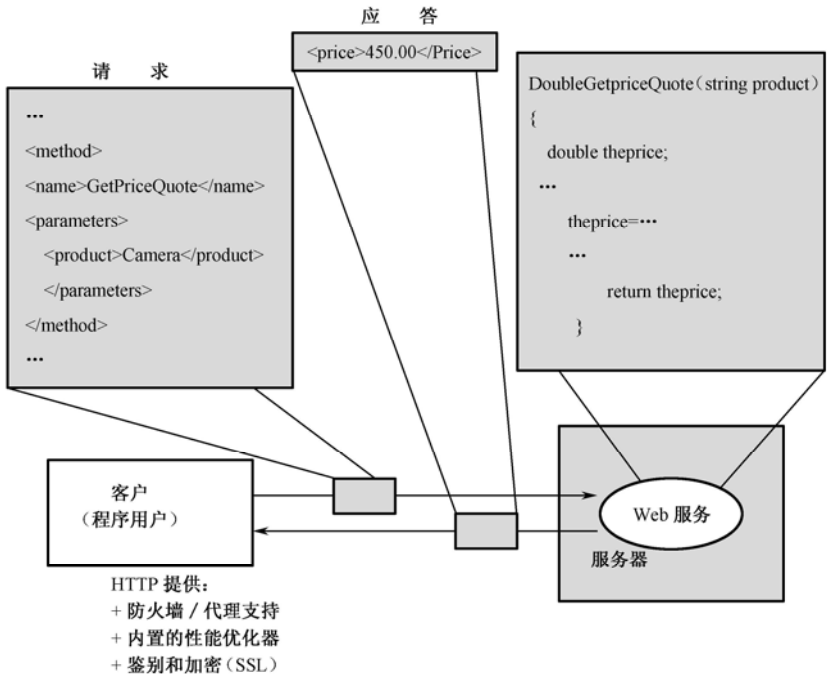


图 8-1 Web 服务特性

对于Web 服务是什么, 以及它如何将得到访问, 现在已经有有了一个初步的认识, 接下来讨论在发布和使用时有的一些基本技术。

8.2 Web服务技术

Web 服务技术并非只有某一家公司极力推崇。实际上，这是许多公司所达成的共识，这中间包括了像 Microsoft 和 Sun 这样的大公司，也包括像 UserLand 这样的不太知名的公司，还有一些如 Ariba 等新近崛起的公司。这种共识已经发展为一组标准，并正在不断发展，如发展的 HTML 标准等。标准化则由 World Wide Web 协会（W3C）来协调。然而，即使基于 W3C 标准，这些实现 Web 服务的技术也存在很大差异，相应的开发机构可以从开放源代码组织到跨国公司，覆盖面很广。

8.2.1 概述

1. Web Service的技术支持简介

Web Service 平台需要一套协议来实现分布式应用程序的创建。任何平台都有它的数据表示方法和类型系统。要实现互操作性，Web Service 平台必须提供一套标准的类型系统，用于沟通不同平台、编程语言和组件模型中的不同类型系统。目前这些协议有：XML、SOAP、WSDL、UDDI，等等。

(1) XML 和 XSD。可扩展的标记语言（XML）是 Web Service 平台中表示数据的基本格式。除了易于建立和易于分析外，XML 主要的优点在于它既与平台无关，又与厂商无关。XML 是由万维网协会（W3C）创建的，W3C 制定的 XML Schema XSD 定义了一套标准的数据类型，并给出了一种语言来扩展这套数据类型。

Web Service 平台用 XSD 来作为数据类型系统。当用户用某种语言如 VB.NET 或 C#来构造一个 Web Service 时，为了符合 Web Service 标准，所有使用的数据类型都必须被转换为 XSD 类型。例如，想让它使用的不同平台和不同软件的不同组织间传递，还需要用某种协议将它包装起来，如 SOAP。

(2) SOAP。SOAP 即简单对象访问协议（Simple Object Access Protocol），它是用于交换 XML 编码信息的轻量级协议。它有 3 个主要方面：XML-envelope 为描述信息内容和如何处理内容定义了框架，将程序对象编码成为 XML 对象的规则，执行远程过程调用（RPC）的约定。SOAP 可以运行在任何其他传输协议上。例如，可以使用 SMTP，即因特网电子邮件协议来传递 SOAP 消息。在传输层之间的头是不同的，但 XML 有效负载保持相同。

Web Service 希望实现不同的系统之间能够用“软件—软件对话”的方式相互调用，打破了软件应用、网站和各种设备之间的格格不入的状态，实现了“基于 Web 无缝集成”的目标。

(3) WSDL。Web Service 描述语言（WSDL）就是用机器能阅读的方式提供一个正式描述文档而基于 XML 的语言，用于描述 Web Service 及其函数、参数和返回值。因为 WSDL 是基于 XML 语言的，所以既是机器可阅读的，又是人可阅读的。

(4) UDDI。UDDI 是一套基于 Web 的、分布式的、为 Web Service 提供的、信息注册中心的实现标准规范，同时也包含一组使企业能将自身提供的 Web Service 注册，以使别的企业能够发现的访问协议的实现标准。通用描述、发现和集成（Universal Description, Discovery, and Integration, UDDI）可以便于 Web 服务的注册和查找。更具体地说，通过它，提供者可以更容易发布可用的功能，而使用者/用户则可以由此找到远程功能。这样，UDDI 的操作就

相当于一个分布式对象系统的注册或命名服务器，它可以使客户不用操心一个服务在哪里，而是提供一种类似于黄页的查找服务。通过与其他技术的联合，它还可以提供每个服务的元数据，这样就扩展了原来的命名服务器概念，即支持对服务的“发现”。

(5) RPC 远程过程调用与消息传递。Web Service 本身其实是在实现应用程序间的通信。现在有两种应用程序通信的方法：RPC 远程过程调用和消息传递。使用 RPC 的时候，客户端的概念是调用服务器上的远程过程，通常方式为实例化一个远程对象并调用其方法和属性。RPC 系统试图达到一种位置上的透明性：服务器暴露出远程对象的接口，而客户端就好像在本地使用这些对象的接口一样，这样就隐藏了底层的信息，客户端也就根本不需要知道对象位于哪台机器上。

操作系统离不开丰富的应用软件的支持。同样，Web Service 这项技术只有通过日益广泛的应用才能体现出其价值。目前比较流行的实现方法是使用.NET 和 Java 两种技术，并且这两种实现方法可以互相操作。如今，可以看到微软、IBM、SUN、Borland 等不同厂商的 Web Service 构建工具建立的 Web Service 应用已经逐渐普及。

2. 综合

所谓的 Web 服务“技术栈”如图 8-2 所示。可以看到，栈中的每一部分都有一个不同的角色。HTTP 和其他传输机制允许数据得到传送。SOAP 是一种与平台无关的方法，可以实现远程服务的调用。WSDL 提供了对服务的灵活声明。最后，UDDI 则允许服务被注册和查找。如图 8-3 所示为使用 Web 服务技术的流程图。

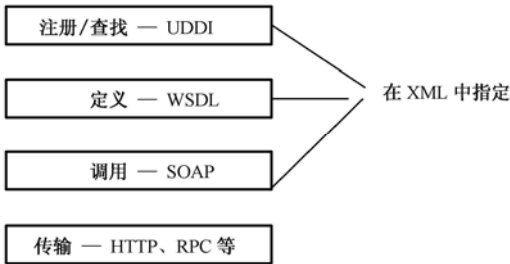


图 8-2 Web 服务技术栈

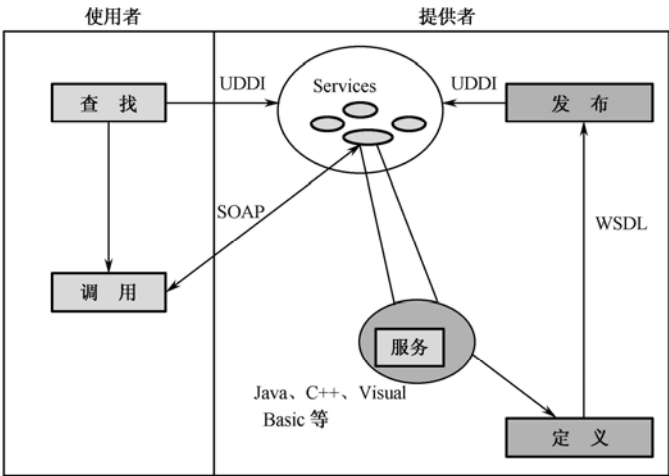


图 8-3 使用 Web 服务技术的流程

使用者首先利用 UDDI 查找或发现一个所需的服务，然后通过发送 SOAP 请求直接与服务器进行交互，而且会通过 SOAP 应答得到回答。在图 8-3 中没有显示各种技术实现的语言，此语言即为 XML。

8.2.2 XML：自描述数据（DTD和模式语言、解析XML）

1. 可扩展标记语言XML

可扩展标记语言 XML 是一种灵活的语言，它允许数据是自描述的。这是标准通用标记语言的一个子集，在这一点上与 HTML 类似。XML 与 SGML 有明显不同，原因是 XML 是一种可使数据建立类型并得到可视化表示的简单方法。而 SGML 是一种更为通用（但同时也很复杂）的元语言，可用于以一种与设备无关并且与系统无关的方式标记文档。另外 XML 与 HTML 也有所不同，因为其可扩展性和重点都基于数据的结构化表示。与此不同，HTML 只有一个有限的标记集，而且重点在于数据的可视化表示。

例 8-1 是一个 XML 文档例子。在此包含了一组待售的老式游戏的信息，其中包括了可用模型和当前价格等数据。

【例 8-1】一个 XML 文档示例。

```
<?xml version= "1.0"?>
  <gamelist>
    <game>
      <name>Gorf</name>
      <model>stand-up</model>
      <price>500.50</Price>
    </game>
    <game>
      <name>Galaga</name>
      <model>cocktail</model>
      <price>1199.99</price>
    </game>
  </gamelist>
```

由这个例子可以总结以下几点：

- (1) 结构：例如，名字、模型和价格都是游戏元素的子元素。
- (2) 可扩展性：这里的标记不是标准集中的一部分，它们要与这个应用的模式相对应。

例 8-1 中的文档可以诊断是“良构”的，因为它满足了 XML 标准的格式化需求。不过，对它是否有效则不明确。为了确保有效性，需要检查在相关模式中这些标记以及其关系是否合法。为此，需要另一种类型的文档——DTD。

2. DTD和模式语言

与 SGML 类似，一个 XML 文档通常与另一个文档相关联，它称为模式定义（schema definition）。模式定义的一般用途是保证与其关文档的有效性，这里主要考虑允许的实体（数据的描述子域）及它们之间的关系。文档类型描述（Document Type Description，DTD）是用于生成模式定义的最通用的模式语言。

例 8-2 包括了一个 DTD，它对前面 XML 文档中老式游戏（例 8-1）的每个元素做了描述。此 DTD 介绍了每个元素，并划分了元素类型，另外还建立了元素之间的关系。

【例 8-2】一个 DTD 示例。

```
<!ELEMENT gamelist(game)+>
<!ELEMENT game(name,model,price)>
<!ELEMENT name(#PCDATA)>
<!ELEMENT model(#PCDATA)>
<!ELEMENT price((#PCDATA))>
```

例 8-2 表达出了以下信息：

- (1) 每个 gamelist 元素可以有一个或多个子 game 元素。加号表示一个或多个。
- (2) 每个 game 元素由 name、mode 和 price 子元素组成。
- (3) 每个 name、mode 和 price 元素包括字符数据（即字符与任何模式元素都没有关系，而只与应用数据有关）。

DTD 可以在一个单独的文件中编码，也可以内嵌到一个 XML 文档中。对于前一种情况，例 8-1 可以调整为例 8-3 代码清单。而对于内嵌 DTD 的情况，在例 8-4 中只需包括例 8-2 代码清单所示的整个 DTD 即可。

【例 8-3】引用一个 DTD 的 XML 文档。

```
<?xml version= "1.0"?>
<!DOCTYPE gamelist SYSTEM "gamelist.dtd">
<gamelist>
    <game>
        <name>Gorf</name>
        <model>stand-up</model>
        <price>500.50</price>
    </game>
    <game>
        <name>Galaga</name>
        <model>Cocktail</model>
        <price>1199.99</price>
    </game>
</gamelist>
```

【例 8-4】嵌入一个 DTD 的 XML 文档。

```
<?xml version= "1.0"?>
<!DOCTYPE gameListDoc[
    <!ELEMENT gameList(game)+>
    <!ELEMENT game(name,model,price)>
    <!ELEMENT name(#PCDATA)>
    <!ELEMENT model(#PCDATA)>
    <!ELEMENT price(#PCDATA)>
]>
<gameList>
    <game>
        <name>Gorf</name>
        <model>stand-up</model>
        <price>500.50</price>
    </game>
```

```

<game>
  <name>Galaga</name>
  <model>cocktail</model>
  <price>1199.99</price>
</game>
</gameList>

```

3. 解析XML

如何解析 XML，已经存在着许多规范和 Java API，最重要的两种 XML 解析技术要算是文档对象模型（Document Object Model，DOM）和简单 XML 解析 API（Simple API for XML Parsing，SAX）。它们都是访问 XML 文档的程序化方法，其区别主要在于前者是 W3C 力推的标准，而后者已经成为事件驱动解析领域事实上的标准接口，这是由 XMLDEV W3C 邮件列表成员合作努力发展而来的。

（1）DOM。文档对象模型（Document Object Model，DOM）是 W3C 所定义的一组对应文档数据的程序化接口，它与平台无关，而且与语言也无关。在 W3C 术语中，“文档”是一个非常通用的概念，HTML 和 XML 文档都可以认为是其子类。

W3C的DOM工作组以级的方式发布了DOM规范。目前有三级规范：DOM 1、2 和 3。DOM3 规范在 2004 年已发布。有关W3C在DOM方面的研究进展，更详细的信息请参见 <http://www.w3.org/DOM/Activity.html>。

DOM 将文档表示为树：根节点有子节点，而每个子节点又有自己的子节点，以此类推。例如，图 8-4 显示的 DOM 即表示了例 8-4 中的 XML。注意元素均表示为方框，而数据则表示为圆圈。

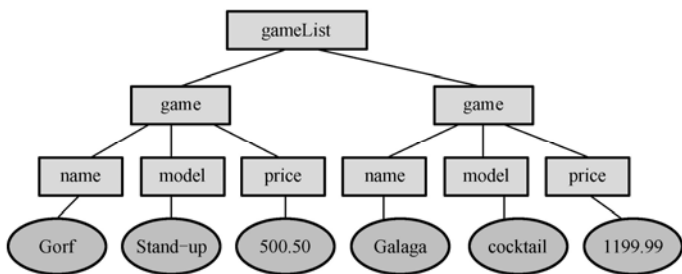


图 8-4 一个 XML 文档的 DOM 表示

将文档表示为树的目的是使程序对文档的访问更加简单，而且可以将面向应用的数据与所有相关元的数据分离开。如果已经将一个 XML 文档解析为一个 DOM 树，就可以访问和更新该文档，在此可以使用任何一种所选定的针对具体语言的 DOM 约束来实现。W3C 发布了抽象 DOM 接口，还发布了面向 Java 和 C++的特定约束。

对于基于 DOM 的 XML 解析/处理，需要记住一个关键问题。如图 8-4 所示，树必须在对文档访问之前创建。这样，一个 XML 文档只能以一种非流文件的方式进行访问，而且必须有足够的本地资源（如内存）用以表示和处理大型的文档。

（2）SAX。SAX 是一个事件驱动的 XML 解析器接口。使用 SAX API 可建立一组“回调函数”，它们将在文档得到解析时被触发。这样，与 DOM 形式的解析不同，在访问文档数据之前，无需等待建立一个树，还可以在解析过程中访问文档元素及其数据。对于从非常大的 XML 文档中抽取数据，这是一个很重要的优点，另外如果希望以一种流的方式处理 XML 方

法时（即文档由一个远程信息源逐步检索），才会体现出其优越性。

没有一个 DOM 树的好处在于，如果只需访问文档中的一部分，那么就不用预留内存来保存文档的整个树。在此，由于数据在 SAX 解析器中流入/流出，内存需求是相对稳定的。在对非常大的文档（可能超出可用内存）进行查询/解析时，此稳定性和资源需求也将成为一大优点。

SAX 形式的解析也有一个缺点，即在解析过程结束时不能访问类似于 DOM 树这样的结构。在元素解析时，可以看到它一次，但也只能如此。除非自行建立了树（或者使用 DOM 重新进行解析），否则无法进行相应的查询。因此，许多人都认为 SAX 解析很适合于处理一次性查询，而 DOM 则更适用于处理多次查询。

注意，SAX 是一个公共域 API，可以在 <http://www.megginson.com/SAX/index.html> 上得到。

（3）使用 Apache Xerces 解析 XML。XML 处理器通常为 Apache Xerces，它既支持 DOM 也支持 SAX API。Xerces 是 Apache 项目所支持的一系列解析器中的最新版本（更多信息请参见 <http://www.apache.org/xml>）。

例 8-5 显示了如何使用 Xerces 来解析所列的 XML，在此使用的是 DOM 解析方法。

【例 8-5】使用 DOM 方法解析。

```
1 import org.apache.xerces.parsers.DOMParser;
2 import org.w3c.dom. Document;
3 import org.w3c.dom.Node;
4 import org.w3c.dom.Element;
5 import org.w3c.dom.NodeList;
6 import org.xml.sax.SAXException;
7
8 import java.io.IOException;
9
10 public class SimpleDom
11 {
12     private Document m_doc;
13
14     public SimpleDom (String a_fileName)
15     {
16         /*Create the parser*/
17         DOMParser dparser =new DOMParser();
18
19         /*Parse the document*/
20         try{
21             dparser.parse(a_fileName );
22             m_doc =dparser.getDocument();
23         }
24         catch (SAXException e) {
25             System.err.println(e);
26             System.exit(-1);
27         }
28         catch (IOException e) {
```

```

29         System.err.println(e);
30         System.exit(-1);
31     }
32 }
33
34 private Document getDocument() { return m_doc; }
35
36 /*Recursively print out element nodes*/
37 private void printNodeAndTraverse (Node node)
38 {
39     /*Print only element nodes*/
40     if ( node.getNodeType() ==Node.ELEMENT_NODE)
41         System.out.println("NODE =" +node.getNodeName());
42
43     /*Call recursively for each child*/
44     NodeList childList =node.getChildNodes();
45     if ( childList != null) {
46         for ( int i=0; i<childList.getLength();i++)
47             printNodeAndTraverse(childList.item(i));
48     }
49 }
50
51     public static void main (String[] args)
52     {
53         /*Create the DOM*/
54         SimpleDom d = new SimpleDom(args[0]);
55
56         /*Traverse all children and print out the name of the nodes*/
57         d.printNodeAndTraverse(d.getDocument());
58     }
59 }

```

这里需要注意的几个主要问题是：

- 第 17 行，解析器在此创建。
- 第 21 行，这里将整个文档首次解析到一个树形数据结构中。
- 第 37 行到第 49 行，对于文档中在此过程输出的元素节点，在此以递归方式进行处理。

如果通过：

```
%java Simple Dom games.xml
```

对以上代码进行编译并运行，则其执行结果为：

```

NODE = gameList
NODE = game
NODE = name
NODE = model
NODE = price

```

```
NODE = game
NODE = name
NODE = model
NODE = price
```

要使用 SAX 解析该 XML 文档，需要利用例 8-6 所示的代码清单实现。

【例 8-6】使用 SAX 方法解析。

```
1  import org.apache.xerces.parsers.SAXParser;
2  import org.xml.sax.Attributes;
3  import org.xml.sax.helpers.DefaultHandler;
4  import org.xml.sax.SAXParseException;
5  import org.xml.sax.SAXException;
6  import java.io.IOException;
7
8  public class SimpleSax
9      extends DefaultHandler
10 {
11     public SimpleSax(String a_file)
12     {
13         /*Create the parser*/
14         SAXParser sparser = new SAXParser();
15
16         /* Set the content handler */
17         sparser.setContentHandler(this);
18
19         /* Parse */
20     try {
21         sparser.parse(a_file);
22     }
23     catch ( SAXException e ) {
24         System.err.println(e);
25     }
26     catch ( IOException e ) {
27         System.err.println(e);
28     }
29 }
30
31 public void startElement (String a_uri,String a_localName,
32     String a_qName,Attributes a_attributes)
33 {
34     System.out.println("NODE = "+a_localName);
35 }
36
37 public static void main (String[] args)
38 {
39     new SimpleSax(args[0]);
```

```
40 }
```

```
41 }
```

此代码中的关键部分为：

- 第 8 行和第 9 行，在此这个 SimpleSax 类扩展了 SAX DefaultHandler 类。
- 第 14 行，在此创建了解析器。
- 第 17 行，在此让解析器了解到是哪一个处理程序（类）在工作。
- 第 21 行，在此开始解析文件（没有立即完成）。
- 第 31 行到 35 行，此为回调函数，对于每个 XML 文档元素都将调用此函数，这里的调用在第 21 行 Parse()调用执行期间完成。

通过比较，可以看到 DOM 虽然比 SAX 代码长一点，但十分简单，可以很容易地想象出执行过程。与此不同，SAX 允许扩展一个处理程序，并可以实现其部分方法（即 startElement()）。而且，由于 SAX 是事件驱动的，这里处理的需求可能被复杂化了。其中，SAX 经常需要保存状态信息的数据结构。例如，如果 XML 解析的目的是计算出在文档中是否至少有两款游戏，那么就必须维护一个全局计数器，使之在每次调用 SAX 回调函数时递增。这个例子虽然很简单，但是由此可以看出，基于 SAX 的处理有时效率高一些，但从本质上讲比基于 DOM 的处理要复杂。

4. XML的特点

- （1）数据的可移植性：使用 DTD，XML 文档可以用任何处理器在任何平台上进行解释。
- （2）提供结构：如层次信息等。
- （3）可读性：与其他二进制编码的文档不同，可以很轻松地看到或设置 XML 文档中的内容。
- （4）可扩展性：可以定义和使用所需的任何模式。
- （5）开放的标准：不属于某一家公司。

这 5 个基本特点就使 XML 成为企业间数据交换的热门语言。XML 提供了 HTML 所缺乏的结构和可扩展性，而且将数据由表示中分离出来。它还很容易跟踪，这与电子数据交换（Electronic Data Interchange，EDI）等难懂的数据交换技术不同。

Web 服务之间的通信以及这些服务的通告都采用 XML 来表示。因此，不仅要理解 XML，还要考虑与其效率相关的问题，这是很重要的。

8.3 用JAX-WS开发Web服务

在学习 Web 服务（Web Services）基本概念的基础上，本节将用 Java 实际开发一个 Web 服务，并介绍其用到的 JAX-WS 工具包。

8.3.1 简介JAX-WS

Java编程语言领域有不少用于开发Web服务的工具包，其中的Apache AXIS工具包是非常受欢迎的开放源码的SOAP实现，AXIS 源自提供开放源码项目的 Apache 软件组织（<http://ws.apache.org/axis>）。

另外一个工具包是基于 XML 的 Web 服务的 Java API（JAX-WS），它是一种开发和实现 Web 服务的 Java API。JAX-WS 处理了 SOAP 层的所有细节，开发者只须调用简单的方法就

可以进行开发，可以从网址 <http://jcp.org/en/jsr/detail?id=224> 得到最新的 JAX-WS 规范。

JAX-WS 是 J2EE 1.4 平台引入的基于 RPC 的编程模型，其功能类似于 JAX-RPC。名称从 JAX-RPC 变为 JAX-WS，主要原因是用到这个类库的应用更多是以 Web 服务为中心的模型。名称的变化可以让这个类库的开发者不必顾及 Web 服务中不再采用的概念，从而更清楚地定义这个库的目的。缺点是使用老版本的库开发的应用必须经过修改才能在新版本下编译通过；新版本的类层次结构发生了很大变化，所以不存在任何兼容的可能性。

为了 Java 应用程序能够和其他使用 SOAP 协议的应用进行通信，新的 JAX-WS 提供了一套 API。JAX-WS 的编程模型的关键在于它通过载入一个 WSDL 文件，或者通过用 `@WebService` (`javax.jws.WebService`) 描述符来标注一个反射类，从而绑定到一个服务上（对应 WSDL 的 `<service>` 节点）。在获得一个服务时，这个程序将请求相应的 `<port>` 并调用所期望的方法。

为了更好地说明客户端程序是如何获取和使用一个 Web 服务的，下面将具体说明如何实际创建一个 Web 服务。第一个 Web 服务是非常简单的 `SimpleService`（类似于一个“Hello, World!”应用）。

8.3.2 下载 CVS 工具

早在学习第 2 章的时候就用到了 JBoss 服务器，现在将要获取一个完整的新版本的 JBoss 服务器。目前这个版本的功能代码仍然在开发中，可以通过一种称为协同版本系统（Concurrent Versioning System, CVS）的工具来得到它们。开发者可以用 CVS 工具保存和管理所有源文件。CVS 服务器跟踪这些文件的变化并且能让开发者了解是谁对文件做了什么样的修改，以及什么时间做的修改。下面将看到，CVS 还可以在已有的源文件上创建不同的分支；源文件中不同的依赖关系可以被注册到一个特定分支上，使开发者在请求时能取得源文件树的“快照”。

第一步是把 CVS 工具下载到相应的平台上。Windows 2000/XP 和 Red Hat Linux（或者任何 RPM 安装系统）的用户可以从 www.march-here.com/cvsnt 找到相应的 CVS 二进制文件。如果使用的是其他操作系统，可以从 www.cvshome.org 得到 CVS 的源代码。习惯于使用 GUI 工具的 Windows 用户可以从 www.wincvs.org 和 www.tortoisecvs.org 得到访问 CVS 服务器的工具。

把 CVS 工具下载到操作系统之后，运行安装包（如果下载的是二进制文件），或者编译源代码后运行安装脚本。打开一个命令窗口（如果还未打开），设置环境变量 `CVSROOT` 为下面的路径：

```
:pserver:anonymous@anoncvs.forge.jboss.com:/cvsroot/jboss
```

Windows 用户通过输入下面的命令来完成设置：

```
>set CVSROOT = :pserver:anonymous@anoncvs.forge.jboss.com:/cvsroot/jboss
```

Linux 和 BSD 系统的用户用下面的命令设置 `CVSROOT` 环境变量：

```
>export CVSROOT = :pserver:anonymous@anoncvs.forge.jboss.com:/cvsroot/jboss
```

设置了正确的 `CVSROOT` 环境变量的，CVS 工具即可知道正确的认证协议、用户名、CVS 服务器和从服务器上获取源文件的初始路径。

下面的命令是希望用 CVS 工具“检出（Check Out）”标签为 `jboss-head` 的源代码文件集：

```
>cvs co -r JBossWS_1_0EA jboss-head
```

其中，`-r` 参数 `JBossWS_1_0EA`（注意这里的 0 是零，不是一个字母）告诉 CVS 服务器

希望获得标签为 `JBossWS_1_0EA` 的代码分支。简而言之，这个 `JBoss` 代码分支包含了 `JBossWS1.0` 的访问版的库，可以用 `Java EE 5` 的 `Web` 服务实现来进行测试。注意这个命令下载了相当大的代码集，所以最好有快速的网络连接和大概 `400MB` 的硬盘空间（编译所有源文件后这个目录将占用大约 `355MB` 空间）。

一旦下载完所有文件，需要把它们编译成一个可以工作的 `JBoss` 服务器。首先转到刚刚下载的源代码目录：

```
>cd jboss-head
```

然后执行相应系统的编译脚本。`Windows` 用户输入下面的命令：

```
>build/build.bat
```

`Linux` 和 `BSD` 用户输入下面的命令：

```
>build/build.sh
```

上面执行的脚本将用 `Ant`（一种帮助简化编译工作的 `Java` 工具，参考 <http://ant.apache.org>）编译下载的整个源代码包。这个脚本成功执行之后，需要转到 `Webservices` 目录：

```
>cd webservices
```

然后运行 `Ant` 编译脚本创建 `JBossWS` 的访问版本：

```
>../tools/bin/ant deploy-jbossws
```

现在已经编译好一个新的 `JBoss` 应用服务器，包括了 `JBossWS` 访问版。余下的工作就是启动这个服务器，转到 `jboss-head/build/output/jboss-5.0.0alpha/bin` 目录下执行 `JBoss` 的“`run`”脚本。现在把 `jboss-head/build/output/jboss-5.0.0alpha` 目录复制到一个更好记的地方，免得每次都使用这个烦琐的路径。本章后面包含代码的路径都指向 `c:\jboss_cvs`，以避免混淆。

8.3.3 创建Web服务

本节介绍第一个 `Web` 服务的例子，并且还没有学习建立和部署 `Web` 服务，所以下面将创建我们自己的代码文件然后再运行它们。首先为这个项目创建一个目录 `JBossWSTest`。接着创建这个例子的 3 个 `Java` 源文件，并把这几个文件简单地放到两个子目录中，分别是 `client` 目录和 `Webservices` 目录。

这个例子需要创建的第一个 `Java` 源文件是服务端点（`endpoint`）接口，文件名为 `SimpleService.java`，客户端和服务端都将用到它。

【例 8-7】 `Java` 源文件 `SimpleService.java`。

```
package webservices;

import java.rmi.Remote;

public interface SimpleService extends Remote
{
    String echo(String input);
}
```

所需的第二个文件是实现了 `Web` 服务接口的类 `SimpleServiceImpl.java`：

【例 8-8】 `Java` 源文件 `SimpleServiceImpl.java`。

```
package webservices;

import org.jboss.annotation.ejb.RemoteBinding;
import org.jboss.ws.annotation.portComponent;
```

```

import javax.jws.webMethod;
import javax.jws.webService;
import javax.ejb.Remote;
import javax.ejb.Stateless;

@WebService(name = "EndpointInterface",targetNamespace = http://localhost,
    serviceName = "SimpleService")
@portComponent(contextRoot = "/jbosswwstest",urlpattern = "/*")
@Remote(SimpleService.class)
@Stateless
public class SimpleServiceImpl implements SimpleService
{
    @WebMethod
    public String echo(String input)
    {
        return input;
    }
}

```

【例 8-9】 用于测试 Web 服务的客户端的文件 SimpleServiceClient.java。

```

package client;

import webservicess.SimpleService;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
import javax.xml.namespace.QName;
import java.net.URL;

public class SimpleServiceClient {
    private static final String_namespace = "http://localhost";
    private static final String_service = "SimpleService";
    private static final String_wsdl =
        "http://localhost:8080/jbosswwstest?wsdl";

    public static void main(String[ ] args) {
        try {
            URL defUrl = new URL(_wsdl);
            //Create the service factory
            ServiceFactory serviceFacotry = ServiceFactory.newInstance( );
            //Load the service implementation class
            Service remoteService = serviceFactory.createService(defUrl,
                new QName(_namespace,_service) );
            //Load a proxy for our class
            SimpleService invoker =
                (SimpleService)remoteService.getPort(SimpleService.class);
            //Invoke our interface for each argument

```

```

        for (int i = 0;i < args.length;i++) {
            String returnedString = invoker.echo(args[i]);
            System.out.println("sent string: " +args[i]
                + ",received string: " +returnedString);
        }
    } catch(Exception e) {
        e.printStackTrace();
    }
}
}
}

```

本节中的 Web 服务有一个方法，它取一个 **String** 作为参数，当把字符串 “back at you” 连接到输入的字符串之后，返回一个组合的 **String**，随后将编译运行这个例子。

现在本 Web 服务有 3 个 Java 源文件，下面将深入探讨这些代码，首先考查服务端点 (end-point) 接口，然后讨论 Web 服务的实现类，最后是 Web 服务的客户端。

1. 服务的端点 (endpoint) 接口

SimpleService 定义的接口是服务的端点接口，客户端和服务端都用到它。

服务端口用这个接口作为无状态会话 **Bean** 的远程接口。用标识符 **@Remote** (**SimpleService.class**) 指定这种关系，这个描述符告诉容器使用 **SimpleService** 类型就好像是在使用由 **@Remote** 描述符标注的接口本身。服务器端还根据服务的端点接口定义的结构产生描述 **SimpleService** 的 **WSDL** 文档。

客户端将用 **SimpleService** 请求一个生成的 “代理 (proxy) 类”，它将透明地调用服务端的 **echo** 方法。也就是说，每次调用 **echo (String s)** 将引发 **JAX-WS** 向服务器端发起一个请求并返回它的响应。

相应地，服务的端点接口只定义了一个方法：

```
String echo(String input);
```

对此可能已经熟悉，但不熟悉的可能是定义接口的方法：

```
public interface SimpleService extends Remote
```

SimpleService 扩展的 **Remote** 接口不同于本书前面的会话 **Bean** 中常见的 **javax.ejb.Remote** 描述符。当客户端请求一个接口实例调用服务器端方法时，**JAX-WS** 利用这个 **Remote** 接口 **java.rmi.Remote** 产生相应的实现代码。

注意 **String echo (String input)** 方法取一个 **String** 作为参数输入并且返回一个 **String**。在过去使用 **JAX-RPC** 时，Web 服务的方法接收和返回的都是 Java 的基本类型。这些 Web 服务参数和返回值是用 **WSDL** 描述的，并且只有那些可以用 **WSDL** 表示的 Java 类型能够用在 **JAX-RPC** 兼容的应用的方法签名中。

本书一开始曾提到 Java EE 5 是 J2EE 1.4 的后续版本，添加了对 **JAX-WS** 的支持，但仍然支持 **JAX-RPC**，还可用 **JAX-RPC** 访问 Web 服务，它支持 Java 全部基本类型及其包装 (wrapper) 类。表 8-1 列出了这些类型和类。

表 8-1 JAX-RPC 参数和返回值支持的数据类型

基本类型	包装类（在 java.lang 中）
Byte	Byte
Short	Short
Int	Integer
Long	Long
Float	Float
Double	Double
Boolean	Boolean

JAX-RPC 另外还支持表 8-2 所示的 Java 类，其中包括许多集合(collection)类，如 ArrayList 和 HashMap。

表 8-2 支持的非基本类型的 Java 类

包（Package）	支持的类
Java.lang	String
	BigInteger
Java.math	BigDecimal
	Date
Java.util	Calendar

除了能使用表 8-1 中列出的基本类型和类之外，JAX-RPC还支持只由其自身支持的数据类型组合而成的类。关于JAX-RPC支持哪些类型的具体情况，请参考JAX-RPC规范（可从<http://java.sun.com/xml/downloads/jaxrpc.html>下载）。

2. Web服务的实现类

这个 Web 服务例子的实现类是 SimpleServiceImpl.java，它实现了 SimpleService 接口，使用如下这些描述符来标注这个类：

```
@WebService(name = "EndpointInterface",targetNamespace =
    "http://localhost",serviceName = "SimpleService")
@PortComponent(contextRoot = "/jbosswstest",urlpattern = "/*")
@Remote(SimpleService.class)
@Stateless
public class SimpleServiceImpl implements SimpleService
```

其中的@stateless描述符简单地将这个类标识为服务器容器的一个无状态会话Bean。当然，不可能只把这个类简单地标注为Web服务并认为它具有正确的功能，将一个类指明为Web服务只是将现有类的功能开放。在这个例子中，如果希望确定这个类持续有效并且可由服务器来管理，这样的功能最好由会话Bean来实现。

前面曾提到，@Remote 描述符告诉服务器容器相应的会话 Bean 的远程接口包含在其他类中。

那么下面这行代码是什么作用呢？

```
@PortComponent(contextRoot = "/jbosswstest",urlpattern = "/*")
```

从上面给出的标识符参数可以看出，这行代码的作用是通知服务器容器如何部署，并且让客户端知道这个 Web 服务。这个 Web 服务将在下面的 URL 上开放它的功能：

```
http://localhost:8080/jbosswstest
```

其中的 `urlPattern` 参数仅仅通知 JBossWS 库去处理 `contextRoot(/jbosswstest)` 中收到的任何 (*) 请求。然而，实际上这个描述符是 JBossWS 库专用的；其他应用服务器的用户可能有类似的功能可以用，但也不完全确定。

上述的 `@WebService` 描述符通过标注该服务的名称和名字空间来完成所赋予这个类的功能。其中的 `name` 参数值是该服务的“端口”（是接口），而 `serviceName` 参数是服务本身的名称（可预知的），`target Namespace` 指令通过向 WSDL（相当于服务的地图）设定一个名称空间，进而帮助客户端请求这个服务。这个服务例子中用的是一个通用的值 `http://localhost`。不熟悉 XML 的开发者不必担心位于 `localhost` 的特定代码。这个字符串的用法很像 Java 中的包（package），目的是为了 avoid 同名元素相互冲突。

3. Web服务的客户端

这个 Web 服务的客户端的代码是 `SimpleServiceClient.java`。其中的 `main()` 方法取一个 `String` 数组作为参量，然后循环读取这些参量并调用 Web 服务的 `echo()` 方法来显示。

前面曾讨论过，客户端将通过生成的代理类与一个 Web 服务进行通信。这个例子代码实例化一个新的 `Service` 对象，并提供给客户端以获取 `SimpleService` 代理实例所需的方法。为了做到这点，它将 `serviceFactory` 对象传递给 WSDL 文档（为 Web 服务自动产生）的地址，并传递给一个 `Qname` 对象用于搜索描述 Web 服务细节的 WSDL 文档。

技巧：`Qname` 是一个“专有名称”，用于在 XML 文档中指定一个节点的引用名称。它可以设定一个节点名（`html`，类似于 `<html>`）、一个与特定名称空间相关联的节点名称（`<html xmlns = "http://www.w3.org/1999/xhtml">`）、一个带有名称空间前缀的节点名称（`<xhtml:html xmlns:xhtml = "http://www.w3.org/1999/xhtml">`）。服务端容器根据 `SimpleServiceImpl` 指定的参数自动产生 WSDL 文档，查找与相同参数匹配的节点。JAX-WS 将用传入的 `Qname` 对象确切地指出哪些 WSDL 文档的元素包含着返回一个有效 `Service` 对象所需要的信息。

这时，你可能会想“请等一下，我不记得有任何自动生成的 WSDL 文档”。你是正确的：这个 Web 服务的 WSDL 文档还没有被自动生成。然而，一旦编译和成功部署了这个 Web 服务，就可以从如下模板的 URL 获得这个 WSDL 文档：

```
http://server.address:port/contextRoot?wsdl
```

回想一下如何在 `SimpleServiceImpl` 的 `@WebService` 描述符中将 `contextRoot` 指定为 `/jbosswstest`，WSDL 文档的地址为：

```
http://localhost:8080/jbosswstest?wsdl
```

一旦获得 Web 服务的一个实例，所要做的就是请求相应的“端口”，把它映射为 `SimpleService` 接口，然后进行如下处理：

```
//Load the service implementation class
Service remoteService = serviceFactory.createService(defUrl,
    new QName(_namespace,_service) );
//Load a proxy for our class
```

```
SimpleService invoker =
    (SimpleService)remoteService.getPort(SimpleService.class);
```

接下来调用这个代理（proxy）类的 echo() 方法，它通过 SOAP 和端点（Endpoint）进行通信。这将调用 Web 服务的实现，通过代理的端点返回值，然后返回给客户端：

```
String returnedString = invoker.echo(args[i]);
```

这个返回值最终赋给 returnedString 变量，然后把它打印输出证明这个 Web 服务完成了整个过程：

```
System.out.println("sent string: " +args[i]
    + ",received string: " + returnedString);
```

8.3.4 构建、测试和运行Web服务

前面已经做好了准备工作，下面将构建、测试和启动这个应用。

(1) 打开一个命令窗口、转到这个应用的 JBossWSTest 目录下。

(2) 将 CLASSPATH 设置为下面的值：

```
>set CLASSPATH= .;
c:\jboss_cvs\server\all\deploy\jbossws.sar\jbossws.jar;
c:\jboss_cvs\server\all\deploy\jbossws.sar\wsdl4j.jar;
c:\jboss_cvs\server\all\deploy\ejb3.deployer\jboss-ejb3x.jar;
c:\jboss_cvs\server\all\deploy\ejb3.deployer\jboss-ejb3.jar;
c:\jboss_cvs\client\jboss-jaxrpc.jar
```

(3) 编译 Webservices 和 client 目录：

```
>javac - d.webservices\*.java
>javac - d.client\*.java
```

(4) 在 Webservices 目录下用编译好的类创建 EJB 3 JAR 文件：

```
>jar cf SimpleService.ejb3 webservices\*.class
```

(5) 把 SimpleService.ejb3 复制到 JBoss 部署目录下：

```
>copy SimpleService.ejb3 c:\jboss_cvs\server\all\deploy\
```

(6) 下面验证这个服务是否已经正确部署。打开一个Web浏览器，输入URL地址 <http://localhost:8080/jbossws>。然后将看到如图 8-5 所示的页面。

```

Welcome to JBossWS

This is JBoss J2EE-1.5 compatible webservice implementation,

◆ View the list of deployed Web Services

```

图 8-5 JBossWS 部署页面

单击“View”链接，显示出当前部署的 JBossWS Web 服务列表，如图 8-6 所示；如果没有出现，那么重复这个部署过程并监控 JBoss 在控制台上输出的部署 Web 服务的消息。如果还有问题，可升级 JBoss 版本。有关 JBossWS 子项目的升级版本的具体信息请参考 JBoss 网站。

Registered Service Endpoints

ServiceEndpointID	ServiceEndpointAddress
SimpleService.ejb3#SimpleService/EndpointInterfaceport	http://prometheus:8080/jbosswstest?wsdl

注意在 ServiceEndPointAddress 一栏下面的链接。在 SimpleServiceClient 类中曾提到这个链接：

```
private static final String_wsdl = "http://localhost:8080/jbosswstest?wsdl";
```

单击该链接，显示出服务器端为 SimpleService 的 Web 服务生成的 WSDL 文档。

(7) 当确定这个 Web 服务已经部署完成之后，在命令行执行下面的命令：

```
>java - classpath.;
c:\jboss_cvs\client\jboss-jaxrpc.jar;
c:\jboss_cvs\client\log4j.jar;
c:\jboss_cvs\client\logkit.jar;
c:\jboss_cvs\client\jbossws-client.jar;
c:\jboss_cvs\client\activation.jar;
c:\jboss_cvs\client\jboss-saaj.jar;
c:\jboss_cvs\client\mail.jar;
c:\jboss_cvs\client\wsdl4j.jar;
c:\jboss_cvs\lib\endorsed\xercesImpl.jar;
c:\jboss_cvs\client\jbossall-client.jar;
c:\jboss_cvs\server\default\lib\jboss-remoting.jar;
c:\jboss_cvs\server\default\lib\javax.servlet.jar
client.SimpleServiceClient this is a test
```

在执行时应当看到下面的输出：

```
sent string: this,received string: this
sent string: is,received string: is
sent string: a,received tring:a
sent string: test, received string: test
```

此时，祝贺你已经利用 Java EE 5 完成了第一个 Web 服务。

8.4 面向服务结构

SOA 是英文 Service-Oriented Architecture，即面向服务结构的缩写。从本质上说，SOA 体现的是一种新的系统架构。在基于 SOA 架构的系统中，具体应用程序的功能是由一些松耦合，并且具有统一接口定义方式的组件（即 Service）组合构建起来的。可以说，SOA 的出现，将为整个企业级软件架构设计带来巨大的影响。

8.4.1 SOA简介

1. SOA基本概念

SOA 是一种架构模型，它可以根据需求通过网络对松散耦合的粗粒度应用组件进行分布式部署、组合和使用。服务层是 SOA 的基础，可以直接被应用调用，从而有效控制系统中与软件代理交互的人为依赖性。

SOA 的关键是“服务”的概念，W3C 将服务定义为：“服务提供者完成一组工作，为服务使用者交付所需的最终结果。最终结果通常会使使用者的状态发生变化，但也可能使提供

者的状态改变，或者双方都产生变化”。

Service-architecture.com 将 SOA 定义为：“本质上是服务的集合。服务间彼此通信，这种通信可能是简单的数据传送，也可能是两个或更多的服务协调进行某些活动。服务间需要某些方法进行连接。所谓服务就是精确定义、封装完善、独立于其他服务所处环境和状态的函数。”

Looselycoupled.com 将 SOA 定义为：“按需连接资源的系统。在 SOA 中，资源被作为可通过标准方式访问的独立服务，提供给网络中的其他成员。与传统的系统结构相比，SOA 规定了资源间更为灵活的松散耦合关系。”

虽然不同厂商或个人对 SOA 有着不同的理解，但是仍然可以从上述的定义中看到 SOA 的几个关键特性：一种粗粒度、松耦合服务架构，服务之间通过简单、精确定义接口进行通信，不涉及底层编程接口和通信模型。

其实，SOA 并不是一种现成的技术，而是一种架构和组织 IT 基础结构及业务功能的方法，SOA 是一种在计算环境中设计、开发、部署和管理离散逻辑单元（服务）的模型。这一定义阐明了 SOA 的范围。

SOA 要求开发人员将应用设计为服务的集合。SOA 要求开发人员跳出应用本身进行思考，考虑现有服务的重用，或思索它们的服务如何能够被其他项目重用。“单独的”、“独立的”、“封装完善的”服务所具有的一个关键的好处是，可以采用多种不同方法将它们组合成较大型的服务，由此来实现重用。

但是，SOA 并不仅仅是一种开发方法，它还具有管理上的优点。例如，现在管理员可直接管理开发人员所构建的相同服务，这远胜于以往管理单个应用的方式。通过分析服务间的交互，SOA 可以帮助企业了解何时以及为什么业务逻辑被切实执行了，这使管理员或分析师能够有针对性地优化业务流程。

2. SOA的特征

(1) 可从企业外部访问。通常被称为业务伙伴的外部用户也能像企业内部用户一样访问相同的服务。业务伙伴采用先进的 B2B 协议（ebXML 或 RosettaNet）相互合作。除了 B2B 协议外，外部用户还可以访问以 Web 服务方式提供的企业服务。

(2) 随时可用。有服务使用者请求服务时，SOA 要求必须有服务提供者能够响应。大多数 SOA 都能够为门户应用之类的同步应用和 B2B 之类的异步应用提供服务。同步应用对于其所使用的服务具有很强的依赖性，但异步应用要更为稳健。

(3) 粗粒度服务接口。粗粒度服务提供一项特定的业务功能，而细粒度服务代表了技术组件方法。例如，向计费系统中添加一个客户是典型的粗粒度服务，而用户可以使用几个细粒度服务实现同一功能。例如，将客户名加入到计费系统中，添加详细的客户联系方式、添加计费信息等。

(4) 分级。一个关于粗粒度服务的争论是此类服务比细粒度服务的重用性差。因为粗粒度服务倾向于解决专门的业务问题，因此，通用性差、重用性设计困难。解决该争论的方法之一就是允许采用不同的粗粒度等级来创建服务。

(5) 松散耦合。SOA 具有“松散耦合”组件服务，这一点区别于大多数其他的组件架构。该方法旨在将服务使用者和服务提供者在服务实现和客户如何使用服务方面隔离开来。

服务提供者和服务使用者间松散耦合背后的关键点是服务接口作为与服务实现分离的实体而存在。这使服务实现能够在完全不影响服务使用者的情况下进行修改。

(6) 可重用的服务及服务接口设计管理。如果完全按照可重用的原则设计服务, SOA 将可以使应用变得更为灵活。可重用服务采用通用格式提供重要的业务功能, 为开发人员节约了大量时间。设计可重用服务是与数据库设计或通用数据建模类似的最有价值的工作。由于服务设计是成功的关键, 因此, SOA 实施者应当寻找一种适当的方法进行服务设计过程管理。

(7) 标准化的接口。近年来出现的两个重要标准 XML 和 Web 服务增加了全新的重要功能, 将 SOA 推向更高的层面, 并大大提升了 SOA 的价值。尽管以往的 SOA 产品都是专有的, 并且要求 IT 部门在其特定环境中开发所有应用, 但 XML 和 Web 服务标准化的开放性使企业能够在所部署的所有技术和应用中采用 SOA。Web 服务使应用功能得以通过标准化接口(WSDL) 提供, 并可以基于标准化传输方式(HTTP 和 JMS), 采用标准化协议(SOAP) 进行调用。

(8) 支持各种消息模式。SOA 中可能存在以下三种消息模式。在一个 SOA 实现中, 常会出现混合采用不同消息模式的服务。这些消息模式有: 无状态的消息、有状态的消息, 等等。

(9) 精确定义的服务接口。服务是由提供者 and 使用者间的契约定义的。契约规定了服务使用方法及使用者期望的最终结果。此外, 还可以在其中规定服务质量。此处需要注意的关键点是, 服务契约必须进行精确定义。

由于 SOA 有上面的特点, 采用 SOA 对企业和开发人员都有好处。例如, 编码更灵活、开发人员角色更明确、支持多种客户类型、系统更易维护、更好的伸缩性、更高的可用性等。

SOA 可以看做是 B/S 模型、XML/Web Service 技术之后的自然延伸。SOA 将能够帮助我们站在一个新的高度理解企业级架构中的各种组件的开发、部署形式, 它将帮助企业系统架构者以更迅速、更可靠、更具重用性架构整个业务系统。较之以往, 以 SOA 架构的系统能够更加从容地面对业务的急剧变化。

3. SOA的起源

要讲清楚 SOA 的起源, 必须涉及 XML 和 Web Service 的起源和发展。本章前面已介绍 XML 和 Web Service 的基本概念及特征, 现在介绍其起源和发展。

(1) XML 的起源与发展。扩展标记语言(XML) 为 W3C 所创建(同 HTML 一样), 源自流行的标准通用标记语言(SGML), 它在 20 世纪 60 年代后期就已存在。这是广泛使用的元语言, 允许组织增加原始文档数据。

XML 不仅被用于以标准化的方式来表达数据, 其语言自身还被用做一系列的附加规范的基础。XML Schema 定义语言(XSD) 与 XSL 转换语言(XSLT) 都以 XML 表达。这些规范事实上已成为关键核心 XML 技术集的关键部分。

XML 表达架构代表了 SOA 的基础层。在其内部, XML 建立了在服务各处流动的消息格式与结构。XSD Schemas 保持消息数据的完整与有效性, 而且 XSLT 使得不同的数据表达间通过 Schema 映射而能够互相通信。

(2) Web Service 的起源与发展。在 2000 年, W3C 接受了一项关于简单对象访问协议(SOAP) 规范的提案。这个规范本来设计用于(并在一些案例替代) 专有 RPC 通信。想法是对于在构件间传输参数数据可以序列化 XML 传送, 然后以序列化生成其原生格式。

很快, 更多的软件厂商开始看到, 对于推进通过构建于专有一免费的互联网通信框架之上的电子商务技术, 存在日益巨大的潜力。这导致了创建一个纯粹的、基于 Web 的分布式技

术能充分利用概念标准化的通信框架，来桥接组织之间和组织内部所存在的巨大差异。这个概念被称为 Web 服务。

Web 服务最重要的部分是其公共接口。它是分配服务识别并使其激活的核心信息块。因此，首先支持 Web 服务的是 Web 服务描述（WSDL）。W3C 第一份 WSDL 评议提案出现在 2001 年，此后还在不断地修订这一规范。

为了进一步地开放协同性的远景，Web 服务需要一个互联网友好的、XML 兼容的通信格式，以便能够建立一个标准化的通信框架。尽管有别的选择，例如，可以考虑 XML-RPC，但 SOAP 因为工业界的偏好而胜出，并且保留了最初的通信标准用于 Web 服务。

Web Service 的概念和 Web Service 的技术支持，详见第 8.1、8.2 节。

4. SOA的出现

后来人们才开始意识到只需要缓和地替代现存的分布式应用，Web 服务可成为独立的架构平台，可使用 Web 服务技术集的效益来实现企业中服务概念的平台。这样，面向服务架构开始进入 IT 行业主流。

在这一点 SOA 频繁地以不同的方式被分类，经常依赖于构建服务所用的实现技术。早期的模型主要从 Web 服务标准初始系列中得到灵感，将 SOA 定义为一个围绕三个基本构件的架构模型：服务请求者、服务提供者与服务注册，如图 8-5 所示。

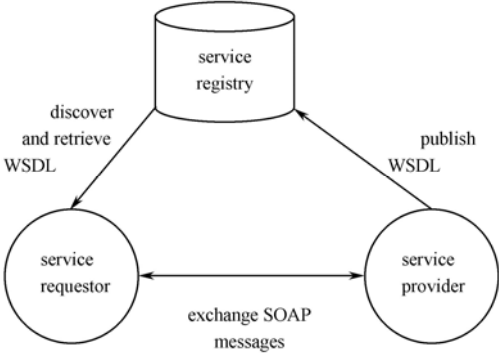


图 8-5 SOA 模型

原始 SOA 模型在今天可轻易获得，因为它已被所有主要厂商的开发及运行平台所支持。这些相同厂商都有关于 SOA 的远大计划，其中许多现在已经能够自我证明。当代 SOA 的诸多特征，大都是过分主动的开发与协作的结果，已经产生了一系列第一代 Web 服务平台的扩展。

8.4.2 SOA的基础架构

顾名思义，服务基础架构（Service Infrastructure）就是以服务为核心建立起来的基础架构。它是一种全新的企业软件类型，帮助用户部署面向服务的架构，使得信息能够在企业内外自由地流动。这一概念由 BEA 公司首先提出，并很快得到全球著名的市场研究机构、软件厂商和企业用户的一致赞同和认可。同时，IBM、Oracle、Microsoft 以及 BEA 本身等国际大厂商都积极投身 SOA 的实践，并努力地推行着。经过几年的发展，已经逐渐形成了一批将 SOA 推向实际应用的成果。

服务基础架构的出现是为了应对大规模实施 SOA 所面临的挑战。它提供了在 SOA 的整个生命周期里部署、配置、保护和管理异构服务所需的全部功能。利用服务基础架构，IT 部

门可以将运行在任何平台（而不仅仅是 Java）上的服务部署在一个共享消息、管理、数据集成和安全服务的基础架构之上；业务流程、安全和数据专家则可以利用现有的服务组合出新的复合应用，无需要求 IT 部门编写新的代码。

对企业来说，“服务基础架构”是实现 IT 与业务同步的关键，它提供了一个能在异构环境中快速、准确、无缝工作的独立平台，可以充分保护客户的 IT 投资，防止厂商垄断，有助于让业务流程、信息和服务在异构的业务环境中安全地流动，并能同时提供专有系统之上的业务逻辑。

今天，SOA 已经成为企业 IT 系统建设的大趋势，越来越多的企业开始实施 SOA。不过，要想真正成功地实施 SOA，就需要一个优化的、开放的服务基础设施，它可以让用户在无需编码的情况下建立起跨异构系统的复合应用。

在经典软件工程理论中，不管是瀑布方法还是原型方法，都是从需求分析做起，一步步构建起形形色色的软件系统。但是需求变更像一个挥之不去的阴影，时刻伴随着系统左右。每个实际应用系统的开发者都饱尝了在系统进入开发阶段、测试阶段，甚至上线阶段遭遇应接不暇的需求变更的极端痛苦。如何解决这一问题？能否来一场软件开发和架构的革命？SOA 的提出，就是被人看成这样的一场革命，其实质就是要将系统模型与系统实现分割开来。

企业管理活动可以形象地被比喻成一个社会网络的沟通与协调，而我们回顾因特网的发展过程，一个重要里程碑就是 ISO(Internet Standard Organization, 国际标准组织)对 OSI(Open System Interconnect, 开放系统互连)七层网络模型的定义。它不但成为以前的和后续的各种网络技术评判、分析的依据，也成为网络协议设计和统一的参考模型。建立七层模型的主要目的是为解决异种网络互联时所遇到的兼容性问题。例如，Novell 网与 NT 网络之间的网络互联。于是通过 ISO 将服务、接口和协议这三个概念明确地区分开来：服务说明某一层为上一层提供一什么功能；接口说明上一层如何使用下一层的服务；协议涉及如何实现本层的服务。这样各层之间具有很强的独立性，互连网络中各实体采用什么样的协议是没有限制的，只要向上提供相同的服务并且不改变相邻层的接口即可。

SOA 在企业管理软件中的应用价值可以等同于上面提供的七层网络模型。一般认为：SOA，面向服务的架构是一个组件模型，它将应用程序的不同功能单元封装成服务(Service)，不同的服务之间通过定义良好的接口进行通信。接口采用中立的方式定义，独立于具体实现服务的硬件平台、操作系统和编程语言。这种具有统一而标准的接口定义（没有强制绑定到特定的实现上）的特征称为服务之间的松耦合。

作为 SOA 的核心要素之一，服务的目的是要实现与另一项服务的远程通信，尤其是要实现数据互享。而 SOA 架构的目的则是要彻底变革 IT 系统的构建方式，由原来的建立专有的单一应用变为建立更为高级和整合的应用，这种应用的显著特点就是充分利用已有的、可以共享和重复使用的功能，也就是服务。

服务是整个 SOA 实现的核心。SOA 指定一组实体（服务提供者、服务消费者、服务注册表、服务条款、服务代理和服务契约），这些实体详细说明了如何提供和消费服务。遵循 SOA 观点的系统必须要有服务，这些服务是可互操作的、独立的、模块化的、位置明确的、松耦合的，并且可以通过网络(UDDI)查找其地址。

SOA 的灵活性将给企业带来巨大的好处。如果把企业的 IT 架构抽象出来，将其功能以粗粒度的服务形式表示出来，每种服务都清晰地表示其业务价值，那么这些服务的顾客（可能在公司内部，也可能是公司的某个业务伙伴）就可以选用这些服务，而不必考虑其后台实现的具体技术。

如图 8-6 所示为一种 SOA 的一种参考模型。

要使 SOA 具有这种灵活性，需要有一系列实现架构的新方法，这是一项艰巨的任务。企业架构设计师必须要变成“面向服务的架构设计师”，不仅要理解 SOA，还要理解 SOA 在具体应用中的表现。在架构实践和最后得到的架构结果之间的区别有可能非常微妙，但却非常关键。所以，SOA 的实现需要借助企业动态建模在企业管理过程中逐步求精，以达到软件与管理的最佳融合。

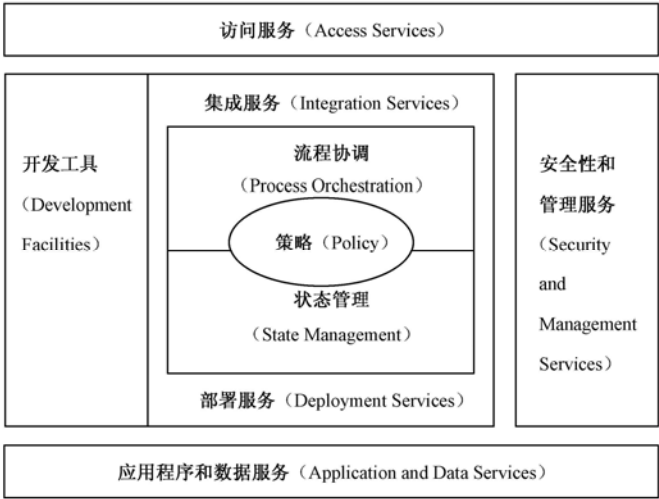


图 8-6 SOA 的一种参考模型

企业动态建模是通过一系列步骤和采用一定的方法，对实际企业对象的发展和变化模式进行分析和简化。去掉对建模目的影响不大的细节现象，得到抽象的动态模型的过程。企业动态建模的过程涉及一系列的活动、方法和工具，用于建立描述企业不同侧面的模型。由于企业组织的复杂性，采用单一的模型无法完整地表达出一个企业多方面的特点，因而在企业动态建模中涉及到多个视角的模型，常用的有 4 个动态模型：信息模型、功能模型、组织模型和流程模型。

从下面的例子中，可以看到 SOA 的应用。

两家大型企业已达成合并协议，其财务系统也随之需要进行整合。让 IT 部门感到高兴的是两家企业使用的财务系统都出自同一厂商，并且是同一产品，因此，IT 部门相信该软件的整合工作会相对快速和简单。然而，工作的最终期限已过去两年，该项目仍然未能完成，预算也超支了 200%。到底发生了什么？

由于企业应用系统包括了多个版本和例程——这大大增加了集成复杂性。两家企业虽然采用了相同的财务系统，但各自的会计业务流程却大不相同，因此分别对应用系统进行了不同的定制。由于定制的业务逻辑隐藏在应用中，因此，进行系统合并时，必须深入了解复杂的底层技术，这需要开发团队耗费数月来编写代码。

应对以上挑战的最佳途径是建立企业级的服务基础架构。服务基础架构的建立使业务逻辑可以在基础架构中进行安全的抽象，并能够采用 XML 实现快速标准化；应用服务被编译为端到端的工作流，而不是系统间脆弱的连接。这将使合并后的企业能够专注于合并业务流程，而不需要再开发数千行的集成代码。

8.4.3 SOA的实现

1. SOA路线图的特征

面向服务的架构是一种 IT 策略，它将企业应用程序中包含的分散功能组织为可互操作的基于标准的服务，这些服务可按照业务需求快速组合和重用。只有平衡了企业的长期目标与短期需求，SOA 的益处才会显现出来。通过在开始采用 SOA 时就指定组织、资金、操作、设计和交付准则，就可保持这一平衡。但这种“大爆炸”式的方法是不可取的，应按照循序渐进的学习曲线，选择一种往复渐进的方式来部署架构更改，这非常重要。大体而言，SOA 路线图就提供了这样一种往复渐进的方式，使用户随着进展得出（重新得出）企业的独有规划。

SOA 路线图应包含下面 3 个关键特征。

（1）成熟：SOA 路线图应该是不断融入经验和教训的“活动文档”。SOA 路线图成熟时，SOA 行动也就以一种可控的方式达到了一个更为精妙的级别。SOA 路线图的创建应该从评估企业当前在 SOA 方面的能力和要求开始。此过程可使用 BEA 的在线自我评估工具作为起点。

（2）作用域：完整的 SOA 路线图应包含 6 个域。

- ① 业务策略与过程：按业务价值排列机会。
- ② 架构：近期、中期、长期参考架构路线图。
- ③ 成本与收益：未来指标、成本构成及收益情况的路线图。
- ④ 构造块：将共享服务战略和标准化进程列入优先地位。
- ⑤ 项目与应用：项目与应用的影响。
- ⑥ 组织与管理：提出的管理结构与策略。

这 6 个域之间有明确的界限，但是仍相互关联、相互依赖。各个域的执行情况是企业级 SOA 行动成功的基石。SOA 路线图应清晰地定义 SOA 行动的边界，并确定一个实现 SOA 目标的明晰、灵活的时限。这些目标应该被分散到多个易于管理的阶段中，随后便可以以一种往复渐进的方式实现。

（3）质量：通过在各里程碑处使用一个“学习与调整”的过程，同时采用往复渐进的方式，路线图将在整个 SOA 行动中保持相关性。为确保 SOA 路线图的质量，应对涉及的应用进行沟通及确认，并向各方征求反馈意见。

2. 构建SOA路线图的步骤

（1）SOA 规划：这一阶段组织并定义 SOA 行动，包括 SOA 的作用域、确定与其他 IT 行动的边界并建立合作、适当地展示 SOA 的业务论证、展示现有业务行动与未来业务行动的衔接关系。

（2）SOA 成熟度评估：在 SOA 成熟度评估阶段，要为当前所处状态建立一个度量标准。此时将定义当前已经实现、可作为 SOA 起点的服务和功能，并确定出可作为基础项目的项目。团队应通过一系列访问调查和问卷调查查看各域，分析、制定基准并验证各域的现状。

（3）SOA 前景展望：团队通过专题研讨会来确定并定义要求的“预期”状态，并确保举办整个企业范围内的联合讨论。

（4）SOA 路线图定义：应该根据前三个阶段所收集的信息，对企业的 SOA 目标和适当的时限进行彻底的差距分析。

3. SOA的实现方案

面向服务架构常常被奉为解决上述业务挑战的一种可行的解决方案。SOA 是一种通过使用和组装构建模块来概念化、设计和构建应用程序的方法，每个构建模块通常被表示为一个可重用的服务。目前使用的许多 SOA 方法只是简单地封装一些业务功能，然后用在应用程序中，而且采用了一种临时、静态和不灵活的方法。开发未来应用程序和业务流程的推荐方法是采用正式的 SOA 实现框架，该框架是动态的、灵活的和可伸缩的，足以满足变化的和复杂的业务需求。不管用户是否为 SOA 实现购买或构建了一个框架，该框架的功能必须要保证有利于用户的解决方案。

SOA 实现框架是一种允许利用 SOA 原理高效构建应用程序和业务流程的技术。它为架构师、开发人员和管理员提供了一个操作框架和工具，允许他们配置、使用和管理企业服务，这些企业服务构成了应用程序和业务流程的构建模块。这个框架在实现流程的各个级别和阶段中使用了一种以服务为中心的方法，并具有以下普遍特征：

- (1) 利用高度安全的、独立于协议的方法来动态连接客户机和服务的能力。
- (2) 可靠的处理服务执行的同步和异步模式的能力。
- (3) 以声明方式定义和处理事件的能力。
- (4) 在客户机和服务器之间动态转换数据格式的能力。
- (5) 以集中方式管理分布式 SOA 资源（服务、配置、策略等）的能力。
- (6) 在服务执行过程中捕获和处理异常的能力。
- (7) 记录和监控在客户服务交易期间出现的不同事件，并进行度量的能力。
- (8) 提供统一的可重用服务调用代码库，用于企业中所有应用程序。
- (9) 支持 Web 服务标准堆栈，以促进大规模的采纳和互操作性。

2005 年 11 月 30 日，IBM、BEA、Oracle、IONA、SAP、Siebel、Sybase、Xcalia 和 Zend 科技公司联合发布了针对 SOA 的编程模型 SCA (Service Component Architecture)、SDO (Service Data Object)，这两个规范的发布标志着 SOA 的实施进入了实质阶段。

这次发布的编程模型中，SDO 相对来说有较长的历史，其版本已经达到了 2.01，相比之下，SCA 则是全新的，其版本仅为 0.9。尽管在此之前，IBM 等厂商在技术上早已经开始支持这两种编程模型，然而对于技术人员来说却很少看到实质性的进展。

2005 年，IBM 曾经基于 SCA 构建了其实现 ESB (Enterprise Service Bus) 的产品，然而该产品毕竟只针对 IBM 公司的 SOA 解决方案，并没有作为一种各厂商都大力支持的 SOA 实现规范。对于一种通用技术来说，如果没有一个实现的标准和规范，其本身是没有太多价值和意义可言的。因此，在经过过去几年的 SOA 理念宣传的阶段后，这项技术已经逐渐走向成熟。

4. 组件体系结构

组件体系结构 (SCA, Service Component Architecture) 可简化使用 SOA 构建的业务应用程序的创建和集成。SCA 提供了构建粗粒度组件的机制，这些粗粒度组件由细粒度组件组装而成。SCA 将传统中间件编程从业务逻辑中分离出来，从而使程序员免受其复杂性的困扰。它允许开发人员集中精力编写业务逻辑，而不必将大量的时间花费在更为底层的技术实现上。

- (1) SCA 方法的优势。

- ① 简化业务组件开发。
- ② 简化作为服务网络构建的业务解决方案的组装和部署。
- ③ 提高可移植性、可重用性和灵活性。
- ④ 通过屏蔽层技术变更来保护业务逻辑资产。
- ⑤ 提高可测试性。

(2) SCA 将构建面向服务的应用程序的步骤划分为两个主要部分：

- ① 实现提供服务和使用其他服务的组件。
- ② 组装组件，以通过服务引用其他服务的方式来构建业务应用程序。

SCA 提供了一种机制，用于打包和部署那些紧密相关的组件，这些组件是作为一个整体开发和部署的。这种机制使服务的实现和组装避免了陷入基础设施功能的细节，也避免了调用外部系统。这样可支持不同基础设施间的服务可移植性。

在 SCA 系统中，SCA 系统用于聚合那些提供了相关业务功能的模块。这是通过配置和管理模块组件、外部服务、入口点，以及连接机制来完成的。SCA 系统的配置由所有部署到其中的子系统的组合加以表示，如图 8-7 所示。

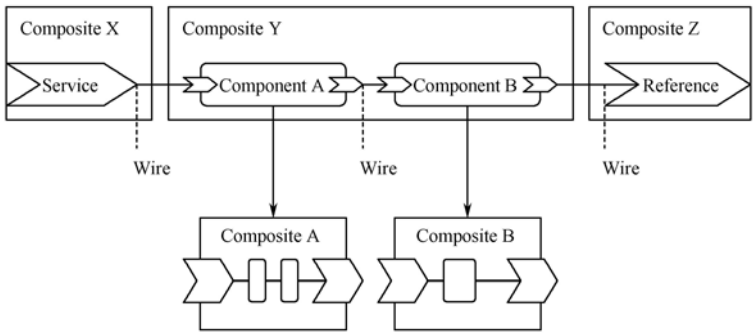


图 8-7 SCA 体系结构

5. 服务数据对象

服务数据对象（SDO，Service Data Objects）是一项新兴标准，用于表示企业应用程序中的数据。SDO 是信息的容器，设计用于提升开放标准和互操作性。SDO 提供了在整个企业应用程序中表示信息的方法，包括表示层、业务逻辑层和此类层之间的通信，如图 8-8 所示。

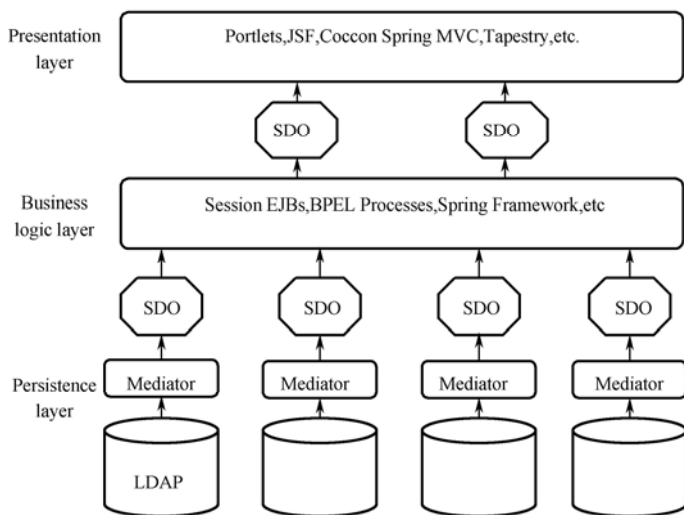


图 8-8 SDO 在系统中的位置

服务数据对象的主要特性有以下几方面。

- (1) SDO 可以包含嵌套对象。此功能称为对象图，是一种非常灵活的表示数据的方式。
- (2) SDO 支持 XPath，可以访问其封装的数据。XML 路径语言 (XPath) 是一项开放标准，是由 World Wide Web Consortium (W3C) 制定的，用于从 XML 文档访问数据。
- (3) SDO 可以作为 XML 构件或 JavaTM 对象存在。借助对 XML 的这项透明支持，直接使用<datagraph>标记作为开头来传递 XML SDO，就可以通过 Web 服务（或任何 XML 传输，如 REST 或 XML-RPC）传递 SDO。
- (4) SDO 包含更改摘要。SDO 更改摘要作为所有活动的历史记录使用。通过使用此功能，应用程序可以将旧数据和新数据区分开。

6. 服务总线技术

服务总线 (ESB, Enterprise Service Bus) 技术采用了“总线”这样一种模式来管理和简化应用之间的集成拓扑结构，以广为接受的开放标准为基础来支持应用之间在消息、事件和服务的级别上动态地互连互通。

ESB 是逻辑上与 SOA 所遵循的基本原则保持一致的服务集成基础架构，它提供了服务管理的方法和在分布式异构环境中进行服务交互的功能。可以这样说，ESB 是特定环境下 (SOA 架构中) 实施 EAI (Enterprise Application Integration) 的方式：首先，在 ESB 系统中，被集成的对象被明确定义为服务，而不是传统 EAI 中各种各样的中间件平台，这样就极大简化了在集成异构性上的考虑。因为不管有怎样的应用底层实现，只要是 SOA 架构中的服务，它就一定是基于标准的。

其次，ESB 明确强调消息 (Message) 处理在集成过程中的作用，这里的消息是指应用环境中被集成对象之间的沟通。以往传统的 EAI 实施中碰到的最大的问题就是被集成者都有自己的方言，即各自的消息格式。作为基础架构的 EAI 系统，必须能够对系统范畴内的任何一种消息进行解析。传统的 EAI 系统中的消息处理大多是被动的，消息的处理需要各自中间件的私有方式支持，如 API 的方式。因此，尽管消息处理本身很重要，但消息的直接处理不会是传统 EAI 系统的核心。ESB 系统由于集成对象统一到服务，消息在应用服务之间传递时

格式是标准的，直接面向消息的处理方式成为可能。如果 ESB 能够在底层支持现有的各种通信协议，那么对消息的处理就完全不考虑底层的传输细节，而直接通过消息的标准格式定义来进行。这样，在 ESB 中，对消息的处理就会成为 ESB 的核心，因为通过消息处理来集成服务是最简单可行的方式。这也是 ESB 中总线（Bus）功能的体现。其实，总线的概念并不新鲜。在传统的 EAI 系统中，也曾经提出过信息总线的概念，通过某种中间件平台，如 CORBA 来连接企业信息孤岛。但是，ESB 的概念不仅仅是提供消息交互的通道，更重要的是提供服务的智能化集成基础架构。

最后，事件驱动成为 ESB 的重要特征。通常服务之间传递的消息有两种形式，一种是调用（Call），即请求/回应方式，这是常见的同步模式。还有一种称之为单路消息（One-Way），它的目的往往是触发异步的事件，发送者不需要马上得到回复。考虑到有些应用服务是长时间运行的，因此，这种异步服务之间的消息交互也是 ESB 必须支持的。除此之外，ESB 的很多功能都可以利用这种机制来实现。例如，SOA 中服务的性能监控等基础架构功能需要通过 ESB 来提供数据，当服务的请求通过 ESB 中转的时候，ESB 很容易通过事件驱动机制向 SOA 的基础架构服务传递信息。

如图 8-9 所示为 ESB 在 SOA 中实现的示意图。

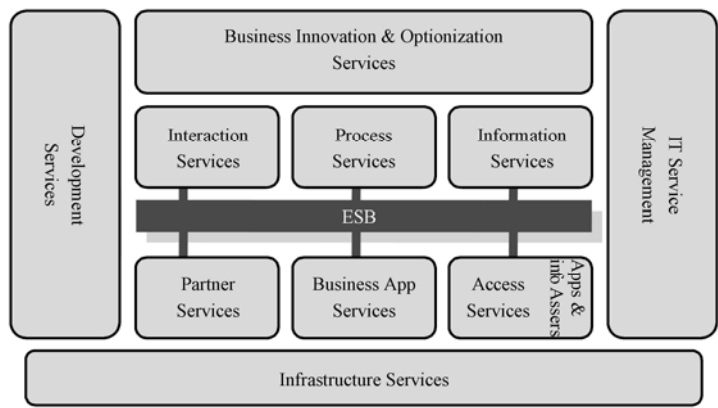


图 8-9 ESB 示意图

8.4.4 SOA 的未来

目前，SOA 在国内外已经成为替代一度风靡的面向对象、BS 结构、三层结构等软件解决方案的一个热门词汇。几乎每一个 IT 公司都有基于 SOA 的解决方案，有支持 Web Service 的产品以及符合 XML 的数据标准。

SOA 的核心价值在于将商业逻辑直接以服务的方式映射到一个服务编排中，从而真正实现商业人员对 IT 系统的直接掌控、修改及再造，一举改变过去很长的 IT 系统建设的内部流程。在此之前，这个系统建设的内部流程包括商业人员提需求、IT 设计师进行模块设计、IT 项目经理组织项目开发、IT 工程师进行开发等。SOA 将改变企业 IT 系统的建设方法及运营方式，将推动软件行业的转型及重新分工。软件行业将逐步形成 3 大类角色：

- （1）应用咨询及集成服务商，负责将服务根据行业最佳实践的方式进行编排。
- （2）服务生产商，负责开发各式各样的服务，并进行登记。
- （3）服务中介，提供服务的注册、查询及搜索。

这样，软件行业将重新组合，形成新的产业链。例如，集成商将最有基础转型为咨询服

务提供商，软件外包公司将最有可能成为服务生产商，而其他公司将面临一个较小的市场空间，为服务中介提供软件系统。因此，集成商要紧紧抓住行业知识库，形成行业咨询能力，而外包厂商要尽快掌握 SOA/Web Service 的技术及相关的标准，其他软件厂商最有可能的发展方向是成为服务中介。这就要求他们完全改变商业模式、人员结构，这将会非常困难。

同时，SOA 将大力推动 BPO（业务流程外包）的运营模式。由于 SOA 对业务流程的充分关注，集成商就能围绕客户的业务流程进行技术及市场工作。因此，这些集成商将处在为客户提供 BPO 服务的有利地位。因此，相信这些集成商将逐步转型为专业的 BPO 提供商。

在国外，SOA 的大规模实施受到传统软件厂商形成的利益团体的阻碍。同时，由于国外信息化建设比较饱和，市场需求乏力，因此，SOA 并没有规模化出现。但是，在标准建设、技术研究以及成熟的工具方面，国外已经有相当的积累。在国内，市场应该能快速接受 SOA 的技术与建设。同时，政府相关管理部门能否看到 SOA 对国内软件产业的革命性影响，将成为能否促使我国成为 SOA 技术的应用大国，从而成为领导世界 SOA 应用发展的重要因素。

SOA 将会使由我国集成商转型成的咨询服务商更有竞争力，因为他们更了解本地的行业特色及具体实践。这些集成商将有机会找到自己真正具有优势的领域，从而摆脱只能处于价值链低端的不利局面。SOA 也将让我国的软件外包公司有一个明确的业务方向及核心技术能力，让我国生产的标准化软件服务畅销世界。

纵观软件发展近 40 年历史，SOA 的发展大致可以分为两个阶段。

第一阶段是以 IBM 为代表、以规模为特征的主机、大系统行业应用软件时代。这个阶段的软件开发是一个独立的大型系统工程。随着网络技术的发展，这种独立开发模式逐步被构件化的软件工程革命替代；第二阶段则是以微软为代表，以技术平台为特征的软件工程年代。在这个阶段，软件开发是一个分工协作的工程建设，并形成了基础软件、平台软件、应用软件、中间件等优化的产业格局。然而，随着软件应用的核心逐步由技术实现转为客户体验。随着互联网的普及与成熟，软件系统的建设将被协同化服务集成替代，从而进入 SOA 的年代。在这个阶段，软件开发、核心技术等不再具有核心价值，快速的服务部署、灵活的流程变化及广泛的服务体系将是重要的竞争优势，而这些特征似乎更加符合我国集成商的特点。

此外，目前在这个领域，还没有看到垄断性的代表力量。因此，SOA 将会推动软件产业以至整个 IT 产业的一次新的结构性变化，将会涌现新的赢家，新的垄断势力，新的技术领袖。

习 题 8

1. 编写一个 Web 服务，读取一个单词并返回这个单词的反向拼写。
2. 编写一个 Web 服务，读取两个数和一个字符串运算符“+”或“-”。然后用这个运算符计算这个数。
3. SOA 的概念是什么？
4. 什么是服务基础架构？
5. 什么是 SOA 的服务？
6. 什么是 SOA 的实现路线？
7. 简述 SOA 和 Web Service 的关系。

第 9 章 Java消息服务等异步技术

本章主要介绍了 Java EE 规范的 Java 消息服务（JMS）的概念、JMS 编程模型、开发消息发布者、开发消息预约者、部署等，以便利用 Java 消息服务来实现异步，基于消息实现高效的系统集成。

本章还介绍了 Java EE 的 Ajax 技术，指出了 Ajax 的核心观念——异步（Asynchronous）与所用到的两个主要技术（JavaScript，XML）。用户填写注册信息时，在不提交到服务器的情况下，判断用户名是否被注册并告知用户。Ajax 的无刷新机制使得注册系统中对于注册名称能即时显示，如果用户名已经存在，则即时通知用户更换名称，提高了系统效率。

9.1 Ajax技术

9.1.1 Asynchronous JavaScript+XML

Ajax 这个名词由 Jesse James Garrett 提出，在他发表的“Ajax:A New Approach to Web Applications”中谈到 Google Suggest 与 Google Maps 所使用到的技术，是他们在 Adivite Path 中称之为 Ajax 的新方法。

在文章中，Ajax 是 Asynchronous JavaScript+XML 的简称，指出了 Ajax 的核心观念（Asynchronous）与所用到的两个主要技术（JavaScript、XML）。

Asynchronous 为非同步，要了解 Ajax，需要先了解为何要非同步。

现在许多应用程序都是在 Web 上建立的。但是，Web 却成为限制 Web 应用程序发展的因素。

限制的原因来自于网络延迟的不确定性，网络连接是耗费资源的行为，程序必须序列化，通信协议沟通、路由传输等动作都很浪费时间和资源。Web 应用程序中，通常通过表进行资料提交，在同步的情况下，使用者发送表单后，只能等待服务器回应，在这段时间内，使用者无法做进一步的操作，如图 9-1 所示。

图 9-1 中加底纹部分是发送表单之后使用者必须等待的时间，浏览器预设是使用同步的方式送出请求并等待回应。

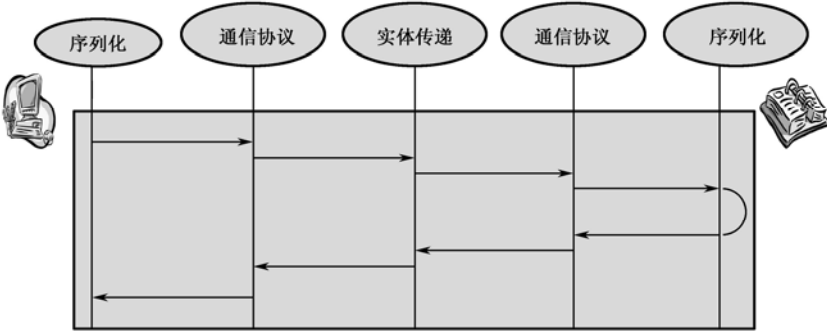


图 9-1 同步技术

如果可以把请求与回应改为非同步进行，即发送请求后，浏览器不需要苦等服务器的回应，而是可以让使用者对浏览器中的 Web 应用程序进行其他的操作。当服务器终于处理完请求并送出回应时，计算机接收到回应，再呼叫浏览器所设定的对应动作进行处理，如图 9-2 所示。

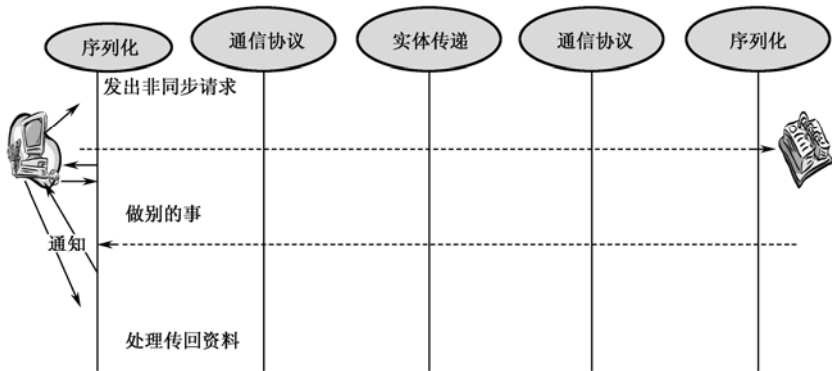


图 9-2 非同步技术

现在的问题是，谁来发送非同步请求？事实上有几种解决方案，在 Ajax 这个名词被提出之间，早就有 Iframe 的方法。

现在谈到 Ajax，都着重在 XMLHttpRequest 组件，可以通过 JavaScript 来建立。其实在 Firefox、NetScape、Opera 中是 XMLHttpRequest，在 Internet Explorer 中是 Microsoft XMLHttpRequest 或 Msxml2.XMLHTTP 的 ActiveX 组件，不过 IE 7 中又更名为 XMLHttpRequest。

Ajax 应用程序是必须由客户端、服务器一同合作的应用程序，JavaScript 用来撰写 Ajax 应用程序客户端的语言，XML 则是请求时或回应时建议使用的交换资料格式。

创建的对象是什么类型、每个创造器的参数在各自的文档中能找到。

9.1.2 XMLHttpRequest

在 Ajax 应用程序中，如果是在 Mozilla/Firefox/Safari 中，可以通过 XMLHttpRequest 来发送非同步请求，如果在 IE6 或者之前的版本，则是使用 ActiveXObject 来发送非同步请求。为了各个不同浏览器的兼容性，必须进行测试取得 XMLHttpRequest 或 ActiveXObject。

【例 9-1】测试取得 XMLHttpRequest 或 ActiveXObject 示例。

```
var xmlhttp;
function createXMLHttpRequest( ) {
    if(window.XMLHttpRequest) { //如果可以取得 XMLHttpRequest
        xmlhttp = new XMLHttpRequest( ); //Mozilla,Firefox,Safari
    }
    else if (window.ActiveXObject) { //如果可以取得 ActiveXObject
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");//Internet Explorer
    }
}
```

在建立 XMLHttpRequest 之后，则可以使用以下几种方法。

- (1) void open(): 开启对服务器的连接。
- (2) void send(): 对服务器传送请求。
- (3) void setRequestHeader(): 为 HTTP 请求设定一个给定的 header 设定值。

- (4) void abort(): 用来中断请求。
- (5) string getAllResponseHeaders(): 传回一个字串。
- (6) string getResponseHeaders(): 传回一个字串。

【例 9-2】一个基本的 Ajax 请求示例。

```
function startRequest( ) {  
    createXMLHttpRequest( );           //建立非同步请求组件  
    xmlhttp.onreadystatechange = handleStateChange;    //设定 callback 函数  
    xmlhttp.open( );                     //开启连接  
    xmlhttp.send(null);                  //传送请求  
}  
  
function handleStateChange( ){          //在这里处理非同步回应  
    ...  
}
```

在 Web 应用中，通常通过 FORM 提交或者连接请求的方式与服务器交互，这种方式总有一个请求和响应的过程，这个过程总是要刷新页面，既浪费网络带宽资源，又影响用户体验。在很多场合需要不断刷新页面，例如，需要连续多次提交请求，这种刷新会严重影响用户的感受。有没有一种方法不刷新页面而完成数据提交或数据请求呢？Ajax 技术就是解决这个问题答案。Ajax 使 Web 应用看上去好像传统窗口应用软件那样立即响应，没有提交、等待、刷新的过程。

Ajax 是利用浏览器与服务器之间的一个通道来“暗中”完成数据提交或者请求的。具体的方法是页面的脚本程序通过浏览器提供的空间完成数据的提交和请求，并将返回的数据由 JavaScript 处理后展现到页面上。整个过程是由浏览器、JavaScript、HTML 共同完成的。Ajax 是这样一组技术的总称。不同的浏览器对 Ajax 有不同的支持方法，而对于 Web 服务器来说没有任何变化，因为浏览器和服务器之间的这个隧道依然是基于 HTTP 请求和响应的，浏览器正常的请求和 Ajax 请求对于 Web 服务器来说没有任何区别。如图 9-3 所示说明了 Ajax 的请求和响应过程。

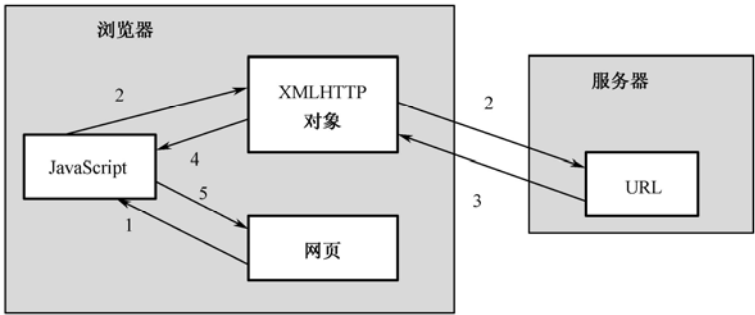


图 9-3 Ajax 的请求和响应过程

Ajax 的请求和响应过程如下。

- (1) 网页调用 JavaScript 程序。
- (2) JavaScript 利用浏览器提供的 XMLHttpRequest 对象向 Web 服务器发送请求。
- (3) 请求的 URL 资源处理后返回结果给浏览器的 XMLHttpRequest 对象。
- (4) XMLHttpRequest 对象调用实现设置的处理方法。
- (5) JavaScript 方法解析返回的数据，利用返回的数据更新页面。

创建的对象是什么类型、每个创造器的参数在各自的文档中能找到。

9.1.3 基于Ajax的用户注册实例

用户填写注册信息时，在不提交到服务器的情况下，判断用户名是否被注册并告之用户。Ajax 的无刷新机制使得注册系统中对于注册名称能即时显示，如果用户名已经存在，则即时通知用户更换名称。

常见的用户注册是用户输入用户名，后台程序检测数据库中用户名是否重复而做出注册成功或失败的提示。这样操作对于用户来说是不方便的。一个好的用户体验应该是：当用户输入完注册用户名后，Web 系统能即时检查并显示，并在检查和显示的同时不影响当前页面的操作。这就是“异步获取数据”的要求，而这是 Ajax 能够完成的。

【例 9-3】基于 Ajax 的用户注册实例。

首先定义 XMLHttpRequest 对象：

```
var xmlhttp = false;
xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
```

然后自定义函数。这个函数的主要功能是异步获得 cu.jsp 的内容。在此之前先提取当前页表元素“u_name”即用户名文本框的值，通过 cu.jsp 其后的参数即赋值而得到不同的结果。

```
function callServer( ) {
    var u_name = document.getElementById("username").value
        //从网页得到用户输入的用户名
    var url = "cu.jsp?name = "+escape(u_name);
    xmlhttp.open("GET",url,true);
    xmlhttp.onreadystatechange = updatePage;
    xmlhttp.send(null);
}
```

cu.jsp 的主要功能就是接收 URL 参数 name 的值做内容显示，该内容最终被 t1.html 异步获取。

```
name = request.querystring("name");
//连接数据库查看是否有该用户，如果有则返回 true，如果没有则返回 false，将异步获取的信息
显示在当前页。
```

```
function updatePage( ) {
    Test1.innerHTML = xmlhttp.responseText;
}
```

9.1.4 Ajax集成技术：DWR

对于程序员来说，现在需要掌握 JavaScript 脚本来操作数据。但是，相对于 Java、JavaScript 语言，无论在面向对象还是数据操作等方面都很弱。

值得高兴的是，针对 Ajax，在 Java EE 领域出现不少解决方案，如 DWR、AjaxAnywhereJSON-RPC-Java 等。

DWR 是开源框架，类似于 Hibernate。借助于 DWR，开发人员无须具备专业的 JavaScript 知识就可以轻松实现 Ajax，使得 Ajax 更加“平民化”。

DWR 的工作原理就是通过把 Java 对象动态地生成 JavaScript 对象，使得客户端通过脚本就能够访问到服务器对象。DWR 大大简化了编写 Ajax 的工作量。

DWR 是一个可以创建 Ajax Web 站点的 Java 开源库。它可以让使用者在浏览器中的 Javascript 代码调用 Web 服务器上的 Java 代码，就像 Java 代码就在浏览器中一样。

DWR 包含两个主要部分：

- (1) 一个运行在服务器端的 Java Servlet，它处理请求并向浏览器发送响应。
- (2) 运行在浏览器端的 JavaScript，它发送请求并能动态更新网页。

DWR 工作原理是通过动态把 Java 类生成为 JavaScript。它的代码就像 Ajax 一样，感觉调用就像发生在浏览器端，但是实际上代码调用发生在服务器端。DWR 负责数据的传递和转换。这种从 Java 到 JavaScript 的远程调用功能的方式使 DWR 用起来有些像 RMI 或者 SOAP 的常规 RPC 机制，而且 DWR 的优点在于不需要任何的网页浏览器插件就能运行在网页上。

Java 从根本上是同步机制，然而 Ajax 却是异步的。所以调用远程方法时，当数据已经从网络返回的时候，要提供有回调（callback）功能的 DWR，如图 9-4 所示。

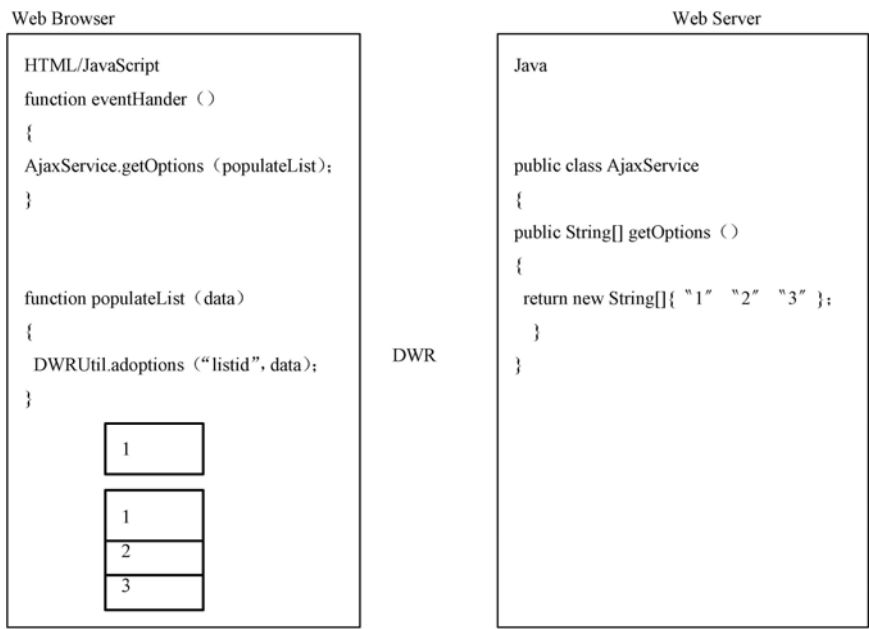


图 9-4 DWR 原理

DWR 动态地在 JavaScript 里生成一个 AjaxService 类，去匹配服务器的代码，由 eventHandler 去调用它。然后 DWR 处理所有的远程细节，包括所有的参数以及返回 JavaScript 和 Java 的值。在示例中，eventHandler 方法调用 AjaxService 的 getOptions()方法，然后通过回调（callback）方法 populateList(data)得到返回的数据，其中 data 就是 String[] {"1", "2", "3"}，最后再使用 DWR utility 把 data 加入到下拉列表。

9.2 Java消息服务概念

9.2.1 什么是Java消息服务

1. B2B型的工作实例

本章中所采用的例子要完成一个简单的任务，即建立一个系统使之可以把产品价格

的更新情况发给多个零售商或分公司。本节所选用的示例公司为 Red Planet 电子公司，它要生产数以千计的电子元件，并分别定价。公司的所有产品价格都保存在一个数据库中，而且 Red Planet 需要对分布在全国的各个零售商实现价格更新。这些更新将逐元件进行，而由 Reb Planet 来管理价格的调整。在 Reb Planet 及其零售商之间的数据流如图 9-5 所示。

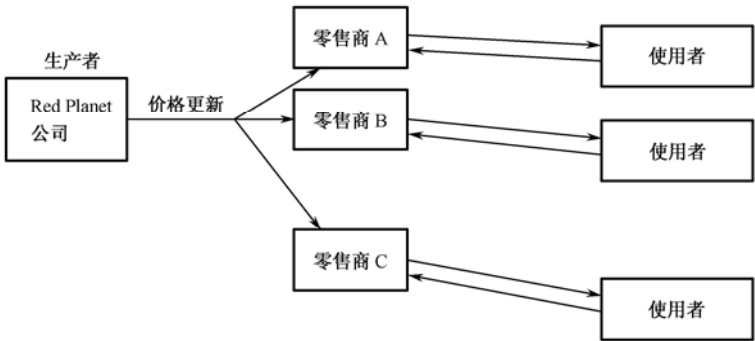


图 9-5 Red Planet 及其零售商之间的关系

零售商对更新的处理分为两个步骤。首先他们要使用 Red Planet 的产品 ID 来查找自己的内部产品 ID，而这可能与 Red Planet 的产品 ID 是相同的，也可能不同（通常零售商要销售多个生产商的产品），同时还会查找出相应产品的毛利，即成本价与售价之间的差价。然后，他们会使用其内部的产品 ID，对内部价格表做相应的更新。这些处理的基本流程如图 9-6 所示。



图 9-6 零售商价格更新流程图

Red Planet 有志于设计一种方案，从而使这种数据传输处理更为高效。我们注意到问题的实质是数据流是单向的，即从一个源到多个目标。还注意到 Red Planet 没有必要处理这些价格更新接收者的应答。公司所希望的就是确保更新最终会送达每个零售商，因此可以稍后再得到相应的确认（若存在确认）。这里的数据传输没有涉及到在线的终端用户；不同于交互式 Web 应用，人本身不会在系统间传输这些价格更新。实际上，整个处理都是通过一个生产应用（Red Planet 的价格更新应用）和下级的使用应用之间的通信完成的。这种解决方案只涉及到两个应用相互通信，而没有任何交互式用户。

性能和可扩展性只是考虑一个异步应用解决方案所需的两个因素。除此之外还有一些其他因素，其中最重要的是便于企业应用的集成。两个应用之间的中间消息服务往往可以加速集成过程，因为它能够使两个系统通过一个公开的 API 很轻松地实现数据传输。更重要的是，基于消息，集成技术人员也不用花太多时间来讨论应用 API 的调整、中间件技术和其他一些问题。

了解了这个示例问题，对于应该采用的解决方案也有了大致的认识，下面就可以深入研究了。不过，在具体到实现细节之前，需要介绍有关 Java 消息服务的一些基本概念，以

及它提供的诸多选择。

2. Java消息服务

Java 消息服务（Java Message Service, JMS）为建立消息解决方案提供了一个与平台无关的 API。与 Java EE 标准的其他部分一样，这并不表示一个具体的解决方案，它只是一个参考 API，消息软件生产商和用户可以将它公开，以便于集成，这样既可以确保灵活性，又能够保证互操作性。JMS 是基于 Java 的，而且 Java EE 还包括了一个参考实现，可以将它作为一个基本的消息提供者者。

对于已经有消息系统的组织来说，JMS 是一种可以提高系统灵活性的方法。不必再依赖于一个特定开发商提供的专用消息 API，这些组织可以使用 JMS 作为所有坚持使用其 API 的开发商的包装器。许多开发商现在也遵循这一标准，因此在选择开发商（或者可能更换开发商）从而使开销降为最小时，就有了更大的灵活性。

对于程序设计人员来说，JMS 提供了一种实现消息的与平台无关的 API，此 API 非常简单而且是抽象的。JMS 支持异步且可靠的消息发送，从而确保消息不仅仅被接收到，而且只能接收一次。JMS 还支持两种抽象消息模型，即点对点模型和发布 / 预约模型，另外它还允许对消息部署进行配置从而保证性能或可靠性，抑或两者兼备。最后，JMS 为与现有 Java EE 技术的集成提供了机会，这些技术包括企业 JavaBean 和 Java 事务 API，等等。例如，可以使用 JMS 技术构建一个分布式异步事务系统。

9.2.2 提供者、客户、消息与管理对象

1. JMS概念

对于 JMS 编程模型，有 4 个基本的键抽象：提供者、客户、消息、管理对象。

如图 9-7 所示，显示了前三者之间的关系。由于管理对象描述的是一组对象，可将它单独地详细加以介绍。

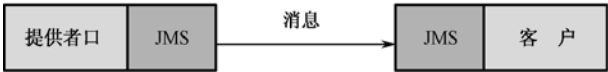


图 9-7 提供者、客户和消息之间的一般关系

尽管非 JMS 专用客户 API 中并不需要包括这三种抽象，但是它们可能通过消息提供者与其他做了此抽象的 JMS 客户进行交互，这里消息提供者除了提供自己的 API 外，还提供了 JMS 兼容性。对于 JMS 客户程序设计人员来说，这一点往往会被忽视。只要提供者支持一个 JMS 接口，JMS 客户就可以与非 JMS 客户透明地实现消息交换。

2. 提供者

JMS 提供者是一个消息系统，它实现了 JMS JPI，而且支持 Java EE 指定的管理和控制特性。从概念上说，提供者要管理消息队列，并协调客户的松散耦合，例如，确保一个消息发送到了多个客户（如果需要的话）。它还可以保证客户接收消息，并为客户接收特定的消息提供了一种方法，这些消息在加入到分布列表之前是分布的。提供者还支持对消息的持久保存，以保证系统失败时的健壮性，同时这种功能还可以实现归档。

3. 客户

JMS 客户可以使用 JMS API 生产或使用消息。客户将得到由提供者控制的对象句柄，如一个消息队列，并用它们来实现消息的分布或对消息进行访问。由于消息是一个对等的概念，因此没有真正意义上的服务器，这里只有客户。在 Red Planet 例子中，无论是总部的系统，还是各个下级使用者（零售商）均为客户。

4. 消息

JMS 消息是生产者与使用者之间传送的基本数据结构。JMS 消息包括一个头、多个属性（可选）和一个消息体（可选）。

（1）JMS 消息头，与 HTTP 消息头类似，包括了有关消息内容的元数据。如表 9-1 所示，列出了 JMS 规范所定义的头字段。

① 消息既可以由提供者发送也可以由客户发出。此处主要考虑的是总部到零售商这条发送路径，不过消息确实可以支持双向通信。注意不要把双向通信与异步混淆。注意，前者表示两个实体之间的通信路径，后者则表示两个实体握手的类型。

表 9-1 JMS 消息头

字 段	由……设置	表 示
JMSDestination	Send 方法	消息的目标
JMSDeliveryMode	Send 方法	持久保存或不保存
JMSExpiration	Send 方法	消息过期前的计划时间
JMSPriority	Send 方法	消息的紧急程度（0~9）。级别 0~4 一般表示正常；级别 5~9 则表示紧急
JMSMessageID	Send 方法	唯一地标识每个消息（对于每个提供者）
MSTimestamp	Send 方法	提供者发送消息的时间
JMSCorrelationId	客户	将一个消息与另一个相关联；例如，客户可以向一个特定请求发出一个响应
JMSReplyTo	客户	提供者应答应该发送到哪
JMSType	客户	参考消息提供者的存储库的定义（JMS 并不提供一个默认的存储库）
JMSRedelivered	提供者	提供者重新发送一个原来未经客户确认的消息

② 许多值实际上要由消息的发送方法来设置。这里所说的“发送方法”是指提供者用于同客户通信的具体方法。如果需要的话，管理者可以重载部分值（JMSDeliveryMode、JMSPriority 和 JMSExpiration）。

（2）消息属性，是一种扩充头中基本信息的可选方法。如果希望扩充表 9-1 所示的消息头来包括其他一些信息（即特定于应用或基础架构的信息），那么消息属性就会很有用。

（3）消息体，包括了实际的业务信息值。JMS 定义了 5 种基本的消息体类型，如表 9-2 所示。

表 9-2 JMS 消息体类型和内容

消息体类型	消息体内容
StreamMessage	与原 Java 类型相关的一串值
ObjectMessage	所有实现 java.io.Serializable 的 Java 对象
TextMessage	一个 Java String
MapMessage	一组名/值对，其名即 Java String 对象，而值为任何原 Java 类型
BytesMessage	一串未解释字节

在 Red Planet 例子中，消息体中包括有一个产品 ID 和新的价格。这样，它无疑要属于 MapMessage 一类（其中名为产品 ID，而值为新价格）。不过，为了使这个例子尽可能简单，使用了 TextMessage 类型。关键是要取决于提供者来确定如何对其发送的数据进行打包，只要属于表 9-2 所列的某一种类型即可。强调可串行化类型的一个原因在于，需要处理一种持久性传送模式。稍后将讨论此模式，还会与非持久性传送模式做比较。

注意：消息体是可选的。例如，有些体为空的消息将作为控制消息来发送，并以此表示一种确认。

5. 管理对象

JMS 管理对象是一些重要的对象，它们构成了客户间通信的基础。管理对象中包括两个重要的子对象：连接工厂和目标。连接工厂（connection factory）可作为一种使 JMS 客户端创建与提供者连接的方法。目标（destination）则是消息生产者的虚拟目标，也是消息使用者的消息源。

比较合适的目标可以是队列（queue）或主题（topic），与这些目标的通信可以使用连接（connection）来实现，而连接则由连接工厂进行分配。一旦在 JMS 使用者和提供者之间建立了一条虚拟的连接，就会创建一个会话（session），并在此生产或使用消息。会话可以使消息生产者和消息使用者被实例化，还为通信提供了一个事务性环境。

9.3 JMS编程模型

9.3.1 两种JMS编程模型

为了理解队列和主题的可应用性，首先需要介绍两个 JMS 所支持的基本编程模型，即点对点模型（point-to-point, PTP）和发布/预约模型（publish/subscribe, pub/sub）。虽然严格地说，它们是消息模型，不过有时也把它们称为域（domain）或消息样式（messaging style）。在 JMS 规范中交替使用这些术语。

设计 PTP 模型的目的是要在一个生产者和一个使用者之间使用。消息被添加到队列中，并被使用者访问。简单地说，在 PTP 通信中只涉及到两方。

相比之下，pub/sub 模型则基于一个主题的概念。发布者要创建主题，并发送相应的消息。使用者则是对应某个特定主题的一个或多个预约者。预约者可以同步或异步接收消息，后者要通过一种消息监听机制实现。

如图 9-8 和图 9-9 所示显示了 PTP 和 pub/sub 消息模型的概念上的区别。

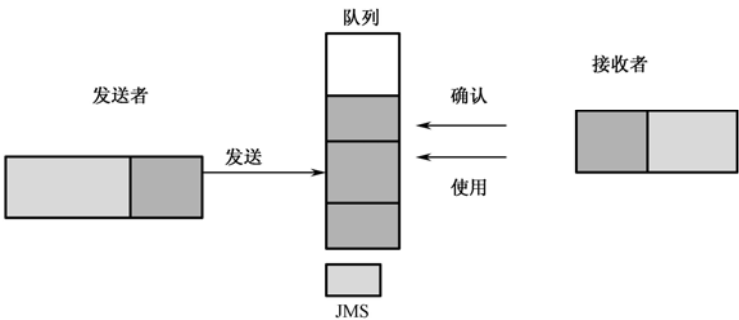


图 9-8 点到点消息模型

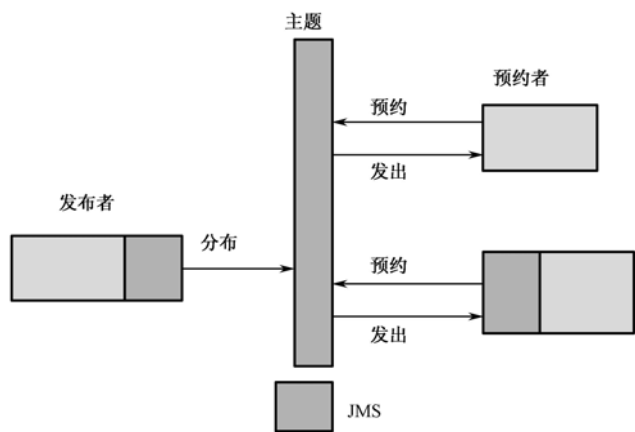


图 9-9 发布/预约消息模型

9.3.2 特定于模型的管理对象接口

JMS 与 Java EE 其他部分一样，只是一个 API，而不是具体实现。它定义的接口集可以分为两个子集：与编程模型（PTP 或 pub/sub）无关的部分，以及与模型相关的部分。一般来说，后者对前者是一个补充，从而更适合于相应的编程模型。相关概念与接口的匹配如表 9-3 所示。

表 9-3 JMS 概念与特定于模型的接口的匹配

与模型无关的接口	PTP 接口	pub/sub 接口
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber
	QueueBrowser	

9.3.3 消息使用的异步性

JMS 有一个重要的也是比较敏感的特性，即需要处理客户的异步性。特别地，将消息发送给客户可能是同步的，也可能是异步的。尽管由生产者向使用者传输数据时是异步的，但从生产者的角度看，只有在必要情况下才需要异步，即生产者要决定何时发送数据，而使用者没有任何关系。不过，尽管使用者从概念上说可以由队列接收消息，但它还要依赖于相应的生产者，即要保证消息确实得以生成。特别地，它不能使用尚未生成的消息，并且必须确定如何发现由提供者生成的新消息。

JMS API 允许消息使用者以下面两种模式执行：

- （1）阻塞模式，其主线程要等待一个新的消息到达。
- （2）非阻塞模式，其主线程继续执行，而提供者线程则在到达一个新对象时执行一个消息监听者方法。

9.4 JMS可靠性与性能

基于所需要的可靠性程度，可以有多种方法来配置 JMS 消息机制，所选择的配置方法会直接影响到系统的性能。

通常情况下，为了性能而对可靠性做出让步是一种提高系统吞吐量的方法。但在应用中，本质上每个消息的使用都很关键，那么就不能忽略 JMS 提供的可靠性控制。

在可靠性和性能之间做出权衡主要有两种方案：客户确认和消息持久保存。

9.4.1 客户确认

JMS 消息发送时，它们可能是一个事务的一部分，也可能不是，也就是说，可能为非事务性的。事务非常重要，因为在事务中允许将一系列消息处理为一个原子工作单元，就像在数据库事务中一样，要么全部成功，要么全部失败。

根据应用的不同，事务可能是必要的，也可能不是必要的。例如，考虑一个银行应用，它要发送两个消息来指示由一个账户向另一个账户实现内部资金转账。如果用户 Joe 在账户 A001 和 A002 之间转账了 100 美元，可将这些消息用伪代码记做：

```
Withdraw(A001,100)
```

```
Deposit(A002,100)
```

在这种情况下，回收资金和资金记入操作要么都被处理，要么都不能处理。否则，如果在资金回收处理完后系统出现故障，那么这个银行客户应付突然丢失 100 美元。在这种条件下，事务是必要的。不过，在我们的 Red Planet 例子中，价格更新是一种原子性操作，因此无须包装在一个父事务中。

如果需要事务，JMS 的确认将自动完成。这是因为一个事务只有在所有操作验证完成之后才能提交。因此，事务的提交就暗含了确认。

不过，如果消息是非事务性的，则有三种 JMS 确认机制可供选择。

(1) **AUTO_ACKNOWLEDGE**：如果客户同步地接收消息，会话会在接收消息的 API 调用返回后自动确认其接收。如果客户异步地接收消息，会话则在对消息监听处理程序的调用返回后自动确认其接收。

(2) **CLIENT_ACKNOWLEDGE**：要由客户显示地通过调用消息对象本身的一个方法来确认消息。客户可以使用此机制周期性地发出确认。假设客户（按顺序）接收到消息 A、B 和 C，但只对 B 做出确认。结果将是前两个消息（A 和 B）被认为得到确认，而 C 未确认。

(3) **DUPS_OK_ACKNOWLEDGE**：这是一种最懒的确认方法，在某些失败的情况下可能会导致重复的消息发送。

9.4.2 消息持久保存

关于消息持久保存，JMS 提供了两种选择。在提供者由消息生产者接收到持久保存消息时，这些消息将保存起来（保存在一个文件或数据库中）。这样，如果提供者在消息得到使用之前的某一点出现问题，这些消息也不会丢失，只要提供者系统恢复回来，消息仍可用。非持久消息则不会在生产时得到保存，因此提供者出现失败时如果没有被使用就会丢失。

持久保存也可以逐消息地指定。这样就为生产者提供了一种异构可靠性机制的灵活性，即大多数消息均为非持久保存的，而将周期的总结进行持久保存。例如，在 Red Planet 例子中，所有单个的价格调整可作为非持久保存消息进行传送，而只对每个月或每周的更新总结进行持久保存。这种机制可以保证一定周期的一致性，而其代价只是使生产者和使用者

感到有些复杂。

需要指出很重要的一点，默认情况下，发送的所有消息都是持久的；编程人员必须显式地指定某个消息不需要持久保存。这对性能有着显著的影响，也正由于此，编程人员可以从可靠性或性能的角度在这两种 **JMS** 消息发送方法中做出选择。非持久保存的消息发送会加快系统的运行，这是毫无疑问的。提供者在发送时（特别是向数据库提交数据时）如果未记录数据，就会得到更好的吞吐量，因为它具有更好的可用性。

但是在选择一个非持久保存的传送模式之前，要记住，这种选择是以可靠性为代价来换取性能的。而且如果消息很重要，而提供者出现问题，那么在性能上取得多大的收益也是没有意义的。这种权衡再次向我们证明了一点，尽管我们的目标是建立更快的可扩展性更好的应用，但是如果需要损失基本的应用需求的话，即使达到了这一目标也是没有价值的。

另外，不应该有这种想法，即在一个非持久保存机制中很难出现消息丢失的情况。实际上，这种机会很多。这取决于发布模型（**PTP** 或 **pub/sub**），还与发布者和预约者所在位置有关，另外还会受到它们面对的资源竞争的影响。

9.4.3 时间依赖性和JMS发布模型

这两种 **JMS** 分布模型对于消息使用的可靠性有着不同的影响。**PTP** 消息模型会在新消息产生时加队列，而且对于每个消息只完成一次出队列的工作（无论存在多少使用者）。这样，生产者就可以确保使用者有机会收到所有消息。不过，为了保证多个客户接收到消息，则必须为每个客户创建一个目标。

相比之下，**pub/sub** 消息模型则允许生产者将消息广播到多个客户，但一般只保证客户在预约有效期间才有机会接收到这些消息。默认机制中，所有在创建预约之前发送或在客户存在故障的情况下发送的消息都无法被接收。若原来的消息没有用，而新消息又不重要时，这就很有意义。例如，如果正在广播不重要的股票报价（是通用的信息，而非当天的交易情况），那么 **pub/sub** 模型则更为适合。

不过，很多情况下仍需要向多个使用者传送可靠的消息。为了达到这个目标，**JMS** 提供了持久性预约。这种预约是指提供者可以向生产者确保使用者接收到某次预约的所有消息，直到消息过期或不再预约为止。每个预约均与一个预约者相关联。为了保证在预约开始前创建的消息得到使用，传送模式必须采用持久保存方式。

9.5 一个JMS pub/sub应用实例

在这一节中，将对编写一个消息应用所涉及的细节内容有更深入的了解。本节不再讨论在不同模型（**PTP** 或 **pub/sub**）和可靠性配置（持久保存传送、非持久保存传送、持久性预约和非持久性预约）条件下如何编写消息应用，仍继续以 **Red Planet** 为例来说明，在此涉及到将价格更新分发给零售商，还需要确定一组特定的 **JMS** 部署选择。采用其他的消息模型或其他可靠性配置进行编码与这里所提供的代码是相似的。

再来考虑这个 **Red Planet** 例子，可以很快做出分析，并得到相应的结论。很明显，消息生产者的角色即为总部的价格应用，而使用者角色则由零售商所使用的应用来充当。这说明价格更新是以一种一对多的方式得到分发的，因此对于 **Red Planet** 来说最自然的传送模型是 **JMS pub/sub**。

需要注意，零售商应用可能会瘫痪，或者出于 **Red Planet** 不能控制的某种原因出现不可

用的情况。价格更新并不是业务的附属产物，它们就是业务本身，因此如果不能接收到消息，就会直接影响到零售商（它会采用一个错误的价格）。为了避免这种情况发生，使用持久性预约来设计这个应用。而且，由于支持 JMS 提供者的主机本身有时可能也会瘫痪，因此消息持久保存是必要的。

9.5.1 开发消息发布者

为了在 pub/sub 模型下发布消息，需要完成以下工作：

（1）创建主题（可以通过程序创建，或利用 Java EE 工具）。

- 取得初始上下文环境（JNDI 发现）。
- 由消息提供者得到一个连接工厂。
- 使用工厂得到一个连接。

（2）使用连接创建一个主题会话。

- 找到主题。
- 为主题会话创建一个发布者对象。
- 发布消息。

1. 创建主题

第一步是在提供者端创建一个主题（在此例中提供者有一个队列）。为了达到这个目的，可以通过 JMS API 来实现，也可以利用通常由 Java EE 开发商提供的工具。为了简化这个例子，选择后一种做法。具体说，在此将考虑如何利用 Sun 的 Java EE 参考实现所提供的工具来创建一个主题。

Sun 提供了一个称为 j2eeadmin 的工具，可用于管理 JMS 队列和主题。以下例子演示了如何使用此工具来创建主题：

```
%j2eeadmin-addJmsDestination PriceUpdatesTopic topic
```

下一步，可以使用这个工具来保证确实创建了主题，在此可以列出所有可用的主题和队列：

```
%j2eeadmin-listJmsDestinations
```

```
JmsDestination
-----
<JMS Destination:jms/Queue,javax.jms.Queue>
<JMS Destination:jms/Topic,javax.jms.Topic>
<JMS Destination:PriceUpdatesTopic,javax.jms.Topic>
```

创建了主题之后，下面来编写生产者的代码，从而让它发布该主题的消息。

2. 编写消息生产者的代码

为了得到与消息提供者的连接，需要一个连接工厂。不过，在得到这个工厂之前，需要先找到提供者资源（因为连接工厂即为管理对象），可以利用 JNDI 和 JMS API 调用来达到这个目的：

```
Context ctx = new InitialContext();

TopicConnectionFactory tConnectionFactory = (TopicConnectionFactory)
    ctx.lookup( "TopicConnectionFactory" );
```

```
TopicConnection tConnection =  
    ConnectionFactory.createTopicConnection();
```

基于 JNDI 属性文件的内容将创建上下文环境。这就有助于生产者找到连接工厂和目标（主题和队列）。

还可以使用上下文来找到特定的主题，而所发布的信息正是关于此主题（即 JMS 目标）：

```
Topic priceUpdatesTopic = ctx.lookup("PriceUpdatesTopic");
```

注意，如果目标尚未创建，若还是按前面所示进行操作，则会在运行时得到一个错误。

一旦有了 TopicConnection，就可以用它来创建实现消息传输的会话上下文环境。

```
TopicSession tSession = tConnection.createTopicSession(false,  
    Session.AUTO_ACKNOWLEDGE);
```

在这个例子中，TopicSession 表示：这个非事务性的机制中确认将自动完成（当提供者调用预约者的 receive 方法时即完成确认）。

最后，可以创建一个 TopicPublisher 对象：

```
TopicPublisher tPublisher =  
    tSession.createPublisher(priceUpdatesTopic);
```

由于已经连接到这个新创建的主题上，因此下面可以开始发送消息了：

```
TextMessage msg = tSession.createTextMessage();  
/*Describe the manufacturers price of product ID CXL43550 as $10.95 */  
msg.setText( "CXL43550:10.95")  
tPublisher.publishMessage(msg);
```

在关于 JMS 消息这节中，本书曾提到可以使用一个 MapMessage 或一个 TextMessage 的类型。还有一个选择是使用更为通用的 ObjectMessage 类型，这样就不再需要客户解析。不过，在此也要权衡：如果使用的是 ObjectMessage 类型，那么所有客户都需要相应的可串行化 Java 类文件，并以此处理其方法和数据结构。而且，随着类文件的发展，还将出现相应的维护和支持问题。

9.5.2 开发消息预约者

创建预约者的工作与创建发布者非常相似。一般地，需要完成以下工作：

- 取得当前上下文环境（可以是一个远程 JNDI 发现）。
- 由消息提供者得到一个连接工厂。
- 使用工厂得到一个连接。
- 使用连接创建一个主题会话。
- 找到主题。
- 选择作为一个持久性预约者或非持久性预约者。
- 选择同步或异步消息处理。
- 宣布已经使用了消息。

对于发布者，首先需要引导预约者，这要通过找到一个初始的上下文环境，得到一个连接并且为消息使用创建一个会话等一系列工作来实现。注意获得一个初始的上下文环境可能涉及到与远程发布者的主机和命名服务器相连接（否则就无法找到主题）。具体如何完成要取决于物理队列的位置以及用于实现消息机制的底层传输机构。

【例 9-4】 远程 JMS 客户示例。


```

1  /*Define remote context information.*/
2  Properties env=new Properties();
3  env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
4  "com.sun.enterprise.naming.SerialInitContextFactory");
5  env.setProperty(Context.PROVIDER_URL,
6  "rmi://publisher.somehost.com:1050");
7
8  /*Get the initial context*/
9  try {
10 jndiContext=new InitialContext(env);
11 }
12 catch (NamingException e){
13 System.out.println("Could not create JNDI"+
14 "context:" +e.toString());
15 System.exit(1);
16 }
17
18 /*Lookup a topic using a connection factory*/
19 try{
20 tConnectionFactory=(TopicConnectionFactory)
21 jndiContext.lookup("TopicConnectionFactory");
22 topic=(Topic) jndiContext.lookup(topicName);
23 }
24 catch (NamingException e){
25 System.out.println("Error during context lookup: "+
26 e.toString());
27 System.exit(1);
28 }
29 /*Create the connection and session */
30 try{
31 tConnection=tConnectionFactory.createTopicConnection();
32 tSession=tConnection.createTopicSession(false,
33 Session.AUTO_ACKNOWLEDGE)';
34 }
35 catch (Exception e ){
36 System.err.println(Error during connection/session creation);
37 e.printStackTrace();
38 System.exit(1);
39 }

```

一旦创建了会话，就可以找到一个主题，预约者可对它进行预约：

```

TopicSubscriber tSubscriber=
tSession.createSubscriber(price UpdatesTopic);

```

以这种方式创建预约者可以得到一个非持久性预约的客户。如果想提高可靠性（在客户端），则可以创建一个持久性预约，而在 **Red Planet** 例子中，就需要这种更可靠的做法。为达到这个目的，需要完成一个稍有不同的 API 调用，并指定一个客户 ID：

```
TopicSubscriber tSubscriber=
```

```
    tSession.createDurableSubscriber(priceUpdatesTopic, "updatesSub");
```

之所以需要指定一个客户 ID，是因为持久性预约只能与单一的一个预约者相关联。如果需要与多个客户建立持久性预约，就需要为它们逐个地创建主题。使用一个持久性预约，我们还需要确保相应客户 ID 与 TopicConnection 得到正确关联。实现这个工作的一种方法是完成相关的 TopicConnection API 调用：

```
tConnection.setClientID("updatesSub");
```

由于此预约是持久性的，预约者应用即使瘫痪了，那么对于瘫痪期间发出的消息，当预约者重新进行监听时，仍能确保这些消息是可用的。

1. 异步消息处理

前面已经讨论过，需要确定消息如何传送到客户：是采用同步还是异步方式呢？在这个例子中，选择了后者，为实现异步传送需要开发一个 JMS MessageListener 对象，并在 TopicSubscriber 对象上进行注册。此监听者将在消息到来时根据需要得到调用。开发一个监听者的需求是它必须实现 MessageListener 接口，特别是 OnMessage()方法。

【例 9-5】 Red Planet 价格更新一个监听者。

```
1  /**
2   *PriceUpdateListener can be used by a subscriber to the
      PriceUpdatesTopic
3   *to process messages asynchronously.
4   */
5   import javax.jms.*;
6
7   public class PriceUpdateListener
8   implements MessageListener
9   {
10    public void onMessage(Message a_mesg)
11    {
12    try{
13    if (message instanceof TextMessage){
14
15        /*Process price update*/
16
17    }
18    else{
19
20        /*Report unexpected message type*/
21
22    }
23    }
24    catch (JMSEException jex){
25
26    /*Handle JMS exceptions*/
27
28    }
```

```

29     catch (Throwable tex){
30
31         /*Handle misc exceptions*/
32
33     }
34 }
35 }

```

然后可以在 `TopicSubscriber` 对象上注册我们的处理程序：

```

updListener=new PriceUpdateListener;
topicSub.setMessageListener(updListener)

```

2. 同步预约处理

`pub/sub` 模型的一大好处在于既可以使用同步方式也可以采用异步方式来使用消息。在 `PTP` 模型中，则没有选择余地，队列使用者通常只能与消息队列保持同步。为了使 `pub/sub` 模型中的预约者同步处理有效的消息，只需阻塞，并通过预约者的 `receive()` 方法等待下一条消息即可：

```
Message m=topicSub.receive();
```

`receive()` 方法还要接收一个参数，从而在一个指定的时间达到后终止（此时间指定为一个毫秒数）。如果没有指定参数，如例 9-2，那么预约者就会一直阻塞，直到下一个消息到达。

9.5.3 关于部署

一般地，真正异步客户后处理是在预约监听处完成的。例如，在监听者代码中，既可以直接通过 `JDBC` 处理更新，也可以连接到一个已有的客户应用上，并由它完成这个更新。同理，对于生产者，可以建立消息发布代码的另一个接口，也可以将此代码集成到应用中。很显然，消息代码并不长，它为应用集成提供了一个很简单的方法。

为了部署我们这个 `Red Planet pub/sub` 应用，接下来要完成两项工作。

（1）启动运行所有预约者应用（或根据需要增加更多的时间），从而使它们开始监听新的消息。

（2）启动运行发布者应用。

完成上述两项工作，消息就可以根据需要发布了。

习 题 9

1. 什么是 Java 消息服务（JMS）？
2. 什么是 JMS 编程模型？简述两种 JMS 编程模型的特点。
3. 简述 JMS 可靠性与性能。
4. 什么是异步消息处理、同步预约处理？
5. 简述一个 JMS `pub/sub` 应用实例的开发、部署。
6. 使用 `Ajax` 更改用户注册模块。

第 10 章 Java EE综合应用实例—— 公文管理信息系统

本章用 Java EE 的 EJB 3.0+JSF 技术开发一个公文管理信息系统，包括最基本的公文查询、添加、修改和删除等功能。在实现过程中，需要使用 EJB 3.0 实现业务逻辑和数据处理层，包括从数据库查询公文、添加、修改和删除公文等业务逻辑。使用 JSF 技术实现表现层，从而与业务逻辑层进行交互。

本章所介绍的公文管理信息系统虽然规模不大，但包括了 Java EE Web 系统最常用、最基本的功能，在此基础上可以开发出更加完善的系统。通过本章的学习，使读者具备独立使用 Java EE 的 EJB 3.0+JSF 进行企业级综合 Web 应用系统开发的能力（掌握系统设计、数据库设计，实现公文列表、查看公文、添加公文、删除公文的能力）。

10.1 公文管理信息系统概述

本章所实现的公文管理信息系统比较简单，包括了最基本的功能：查询公文、添加公文、修改公文、删除公文和查看公文等，其中查询公文时进行分页显示。

该系统运行环境如下：

- JDK 1.5 开发工具包。
- 服务器：JBoss 5 应用服务器。
- 数据库：MySQL 5 数据库管理系统。
- 浏览器：IE6 浏览器。

管理员登录成功后进入后台可以实现添加公文、修改公文、删除公文等操作。如图 10-1 所示为本系统的功能模块划分。

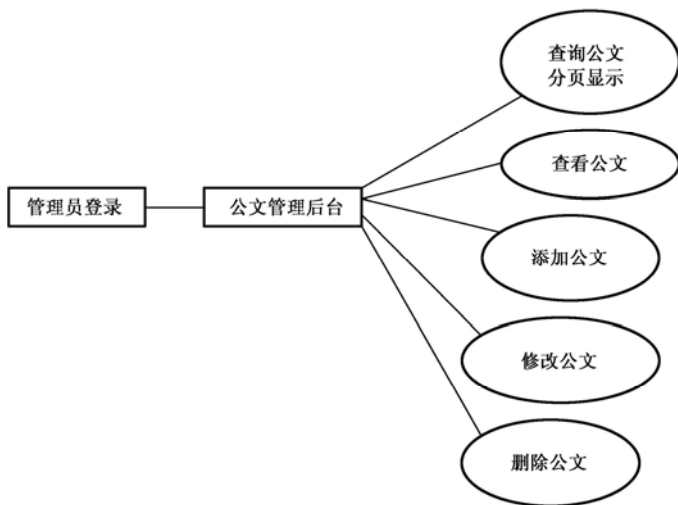


图 10-1 系统功能模块划分

10.2 设计数据库

本实例采用了 MySQL 5 作为数据库，先建立一个名字为 ofiledb 的数据库，下面是该数据库中的表信息。

1. 管理员信息表

管理员信息表（admin）包含管理员 id、登录名称和登录密码三个字段，如表 10-1 所示。

表 10-1 管理员信息表

字段名称	字段类型	是否为空	含义
id	自动编号	否	管理员 id
name	varchar(10)	否	登录名称
pwd	varchar(10)	否	登录密码

在表 10-1 中初始化一行记录，如表 10-2 所示。

表 10-2 初始化记录

id	name	pwd
1	admin	admin

2. 公文类别表

公文类别表（category）包含公文类别 id、公文类别名称两个字段，如表 10-3 所示。

表 10-3 公文类别表

字段名称	字段类型	是否为空	含义
id	自动编号	否	公文类别 id
name	varchar(20)	否	类别名称

在该表 10-3 中初始化一些记录，如表 10-4 所示。

表 10-4 初始化记录

id	name
1	行政公文
2	党务公文
3	教务公文
4	科技公文

3. 公文信息表

公文信息表（ofile）包含公文 id、公文标题、公文内容、公文类别 id、公文时间五个字段，如表 10-5 所示。

表 10-5 公文信息表

字段名称	字段类型	是否为空	含义
id	自动编号	否	公文 id
title	varchar(50)	否	公文标题

续表

字段名称	字段类型	是否为空	含义
content	varchar(200)	否	公文内容
cid	int	否	公文类别 id
time	datetime	否	公文时间

本实例所用的 MySQL 数据库的登录名称是 root，登录密码是 19480425，在进行数据库连接时要注意。

10.3 系统公共配置

前面已经介绍了公文管理信息系统的功能设计及数据库设计，现在可以进行具体的编码工作。首先实现该实例最基本的、通用的公共代码部分，包括一些类库的导入、web.xml 的配置、数据源的配置等。

10.3.1 导入相关类库

由于本实例主要采用 EJB 3.0 和 JSF 进行开发，所以要先将 EJB 3.0 和 JSF 的相关类库导入到实例中。同时由于本实例采用的数据库是 MySQL，所以也需要将 MySQL 数据库的连接驱动导入到实例中，也就是把所需的类库文件复制到项目的 WEB-INF\lib 文件夹中，JBoss 服务器一般都带有这些类库。例如，jsf-impl.jar、jsf-api.jar、commons-digester.jar、commons-collections.jar、commons-beanutils.jar、jstl.jar、standard.jar，都是 JSF 所需的类库。EJB 和数据库连接所需的类库有：javaee.jar、mysql-connector-java-3.1.13-bin.jar。

10.3.2 配置web.xml

导入 JSF 的类库后，还需要在 web.xml 文件中进行配置才能在实例中应用 JSF，代码如下：

```
<?xml version = "1.0" encoding = "utf-8"?>
<web-app xmlns = "http://java.sun.com/xml/ns/j2ee"
xmlns:xsi = " http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation = "http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
version = 2.4">
<context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>/WEB-INF/faces-config.xml </param-value>
</context-param>
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>
        javax.faces.webapp.FacesServlet
    </servlet-class>
</servlet>
<servlet-mapping>
```

```

    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.faces</url-pattern>
</servlet-mapping>
</web-app>

```

10.3.3 数据源配置

本实例采用 JBoss 服务器，所以需要在服务器上配置一个数据源。在 JBoss 安装目录下的\server\default\deploy 文件夹中新建一个名字为 mysql-ds.xml 的文件，代码如下：

```

<?xml version = "1.0" encoding = "UTF-8"?>
<datasources>
<local-tx-datasource>
    <jndi-name>JEE_MySqlDS</jndi-name>
    <connection-url>jdbc:mysql://3306/ofiledb?useUnicode=true&
        characterEncoding=utf-8
    </connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>root</user-name>
    <password>19480425</password>
    <exception-sorter-class-name>org.jboss.resource.adapter.jdbc.vendor.
        MySQLExceptionSorter
</exception-sorter-class-name>
<metadata>
    <type-mapping>mySQL</type-mapping>
</metadata>
</local-tx-datasource>
</datasources>

```

上述标签在前面已经介绍过，这里就不再介绍了。

10.3.4 配置persistence.xml文件

由于本实例使用 EJB 3.0 进行业务逻辑层的开发，所以需要通过 persistence.xml 文件来定义持久化单元，代码如下：

```

<?xml version = "1.0" encoding = "UTF-8"?>
<persistence xmlns = "http://java.sun.com/xml/ns/persistence"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation = "http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"version = "1.0">
<persistence-unit name = "JEE_ofile_PU"transaction-type = "JTA">
    <jta-data-source>java:/JEE_ofile_MySqlDS</jta-data-source>
    <properties>
        <!--调整 JDBC 抓取数量的大小: Statement.setFetchSize() -->
        <property name = "hibernamte.jdbc.fetch_size"value = "18"/>
        <!--调整 JDBC 批量更新数量-->
        <property name = "hibernamte.jdbc.batch_size"value = "10"/>
    </properties>
</persistence-unit>
</persistence>

```

```

<!-- 显示最终执行的 SQL- ->
<property name = "hibernamte.show_sql" value = "true"/>
<!-- 格式化显示的 SQL- ->
<property name = "hibernate.format_sql" value= "true"/>
</properties>
</persistence-unit>
</persistence>

```

10.4 公文管理信息系统业务逻辑和数据处理层的实现

本实例的业务逻辑和数据处理层是由 EJB 3.0 实现的。根据数据库的设计，需要实现 3 个实体 Bean 和与之相关的 3 个会话 Bean。这 3 个实体 Bean 分别与数据库中的 3 个表建立了映射关系，下面进行详细介绍。

10.4.1 admin 表实体和对应会话 Bean

与 admin 表建立映射关系的实体是 Admin 类，Admin 类主要完成持久化管理员信息，即需要在该类中创建和 admin 表字段对应的属性，如 name、id 和 pwd 等，并在下面创建相应的 set 和 get 方法设置和获取变量值，代码如下：

```

package huizhi.Entitybean;
import java.io.Serializable;
import javax.persistence.*;
@SuppressWarnings("serial")
@Entity
@Table(name = "admin")
public class Admin{
    private int id;
    private String name;
    private String pwd;
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    @Column(nullable = false)
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}

```



```

@Column(nullable=false)
public String getPwd() {
    return pwd;
}
public void setPwd(String pwd) {
    this.pwd=pwd;
}
}

```

该类包含 id、name、pwd 三个属性，与 admin 表中的字段一一对应。下面实现会话 Bean，首先介绍会话 Bean 的远程接口和本地接口。

```

package huizhi.Sessionbean;
public interface AdminRemote{
    public Boolean checkLogin(String name,String pwd);
}

```

这是远程接口，该接口只包含一种方法。下面是本地接口的代码：

```

package huizhi.Sessionbean;
public interface AdminLocal extends AdminRemote{
}

```

通过上述代码看到本地接口直接继承了远程接口。下面介绍实现上述两个接口的会话 Bean，代码如下：

```

package huizhi.Sessionbean;
import javax.ejb.*;
import javax.persistence.*;
@Stateless
@Remote(AdminRemote.class)
@Local(AdminLocal.class)
public class Adminbean implements AdminRemote,AdminLocal{
    @PersistenceContext(unitName= "JEE_ofile_PU")protected EntityManager em;
    public Boolean checkLogin(String name,String pwd)
    {
        Query q=em.createQuery("from Admin a where a.name =?1 and a.pwd =?2");
        q.setParameter(1,name);
        q.setParameter(2,pwd);
        if(q.getResultList().size() == 0)
            return false;
        else
            return true;
    }
}

```

该会话 Bean 实现了接口中的 checkLogin 方法，该方法用来检验管理员的登录信息，登录成功返回 true，登录失败则返回 false。

10.4.2 category表的实体和会话Bean

与 category 表建立映射关系的实体是 Category 类，Category 类主要用于存储公文类别

信息。**Category** 类需要创建与表 **category** 中字段相对应的属性和方法，代码如下：

```
package huizhi.Entitybean;
import java.io.Serializable;
import javax.persistence.*;
@SuppressWarnings("serial")
@Entity
@Table(name= "category")
public class Category implements Serializable{
    private int id;
    private String name;
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    @Column(nullable=false)
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name=name;
    }
}
```

该类包含 **id**、**name** 两个属性，与 **category** 表中的字段一一对应。下面实现会话 **Bean**，首先介绍会话 **Bean** 的远程接口和本地接口。

```
package huizhi.Sessionbean;
import java.util.ArrayList;
import huizhi.Entitybean.Category;
public interface CategoryRemote {
    public ArrayList<Category>getCategoryList() ;
}
```

这是远程接口，该接口包含一个方法。下面是本地接口的代码：

```
package huizhi.Sessionbean;
public interface CategoryLocal extends CategoryRemote{
}
```

通过上述代码看到本地接口直接继承了远程接口。下面介绍实现上述两个接口的会话 **Bean**，代码如下：

```
package huizhi.Sessionbean;
import huizhi.Entitybean.Category;
import java.util.ArrayList;
import javax.ejb.*;
import javax.persistence.*;
```

```

@Stateless
@Remote(CategoryRemote.class)
@Local(CategoryLocal.class)
public class Categorybean implements CategoryRemote,CategoryLocal{
    @PersistenceContext(unitName= "JEE_ofile_PU")protected EntityManager em;
    @SuppressWarnings("unchecked")
    public ArrayList<Category>getCategoryList()
    {
        Query q= em.createQuery("select c form Category c");
        ArrayList<Category>CategoryList=(ArrayList<Category>)
            q.getResultList() ;
        return CategoryList;
    }
}

```

该会话 Bean 实现了接口中的 getCategoryList 方法，该方法查询 category 表，并把该表中的信息封装成 Category 类存储在 ArrayList 中。

10.4.3 ofile表的实体和会话Bean

与 ofile 表建立映射关系的实体是 Ofile 类，Ofile 类用来持久化相关的公文信息。该类需要创建与 ofile 表中字段对应的属性和方法。代码如下：

```

package huizhi.Entitybean;
import java.io.Serializable;
import javax.persistence.*;
@SuppressWarnings("serial")
@Entity
@Table(name= "ofile")
public class ofile implements Serializable{
    private int id;
    private Category category;
    private String title;
    private String content;
    private String time;
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public int getId {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    @ManyToOne
    @JoinColumn(name= "cid",referencedColumnName= "id")
    public Category getCategory() {
        return category;
    }
}

```

```

public void setCategory(Category category) {
    this.category = category;
}
@Column(nullable=false)
public String getTitle() {
    return title;
}
public void setTitle(String title) {
    this.title = title;
}
@Column(nullable = false)
public String getContent() {
    return content;
}
public void setContent(String content) {
    this.content = content;
}
@Column(nullable=false)
public String getTime() {
    return time;
}
public void setTime(String time) {
    this.time = time;
}
}

```

注意：由于 ofile 表通过该表的 cid 字段与 category 表的 id 字段进行了外键关联，所以在 Ofile 实体 Bean 中对该 Bean 的 category 属性设置“多对一”的映射。下面实现会话 Bean，首先介绍会话 Bean 的远程接口和本地接口。

```

package huizhi.Sessionbean;
import java.util.ArrayList;
import huizhi.Entitybean.Ofile;
public interface ofileRemote{
    public ArrayList<Ofile>getOfileList() ;
    public ofile getOfileByID(int id);
    public void addOfile(Ofile n);
    public void editOfile(Ofile n,int id);
    public void delOfile(int id);
}

```

这是远程接口，该接口一共包含了 5 种方法。下面是本地接口的代码：

```

package huizhi.Sessionbean;
public interface ofileLocal extends ofileRemote{
}

```

通过上述代码看到本地接口直接继承了远程接口。下面介绍实现上述两个接口的会话 Bean，代码如下：

```

package huizhi.Sessionbean;

```

```

import huizhi.Entitybean.*;
import java.util.ArrayList;
import javax.ejb.*;
import javax.persistence.*;

@Stateless
@Remote(OfFileRemote.class)
@Local(OfFileLocal.class)
public class ofFileBean implements ofFileRemote, OfFileLocal {
    @PersistenceContext(unitName= "JEE_ofFile_PU")protected EntityManager em;
    public void addNew(OfFile n){                //添加公文
        em.persist(n);
    }
    public void delOfFile(int id) {                //删除公文
        ofFile n = em.find(OfFile.class,id);
        if(n!= null)
            em.remove(n)
    }
    public void editOfFile(OfFile nn,int id) {        //修改公文
        ofFile n = em.find(OfFile.class,id);
        n.setTitle(nn.getTitle() );
        n.setCategory(nn.getCategory() );
        n.setContent(nn.getContent() );
        n.setTime(nn.getTime() );
    }
    public ofFile getOfFileByID(int id) {            //根据 id 查询公文
        Query q = em.createQuery("select n from ofFile n where n.id = ?1");
        q.setParameter(1,id);
        ofFile n = (OfFile)q.getSingleResult() ;
        return n;
    }
    @SuppressWarnings("unchecked")
    public ArrayList<OfFile>getOfFileList() {        //查询公文
        Query query=em.createQuery("select n from ofFile n order by n.time desc");
        ArrayList newList = (ArrayList<OfFile>)query.getResultList() ;
        return ofFileList;
    }
}

```

该会话 bean 实现了接口中的 5 种方法，这 5 种方法分别实现了查询公文、添加公文、修改公文和删除公文和查看公文等功能。

10.5 公文管理信息系统表现层的实现

本实例的表现层由 JSF 实现。表现层的每个模块基本上都包含三个部分：视图页面、Backing bean 和 faces-config.xml 配置。视图页面提供数据的输入和输出。Backing bean 接收

参数后调用业务逻辑层的本地接口中的方法完成业务处理，并返回相应的信息。faces-config.xml 文件用来配置 Backing bean 和页面导航规则。

10.5.1 登录页面

在实现具体页面中，需要使用 JSF 标签来实现相关布局，当然也可以在 JSP 页面中嵌入 CSS 和 JavaScript 代码。在进入系统之前，需要进行用户登录。当用户验证通过后才可以进行下一步操作，代码如下：

```
<%@page language = "java" pageEncoding = "UTF-8"%>
<%@taglib uri = "http://java.sun.com/jsf/html" prefix = "h"%>
<%@taglib uri = "http://java.sun.com/jsf/core"prefix = "f"%>
<html>
<head>
  <title>后台登录</title>
</head>
<body>
  <f:view>
    <center><h3>后台登录<h3></center>
    <h:form>
      <center>
        登录名称:<h:inputText value = "#{Loginbean.name}"
        required = "true"></h:inputText>
        <br><br>
        登录密码: <h:inputSecret value= "#{Loginbean.pwd}"required= "true"
        style = "width:156px"></h:inputSecret><br><br>
        <center><h:commandButton value = "登录"
        action = "#{Loginbean.checkLogin}"></h:commandButton><br><br>
        <h:outputText value = "#{Loginbean.msg}"
        style = "color:#FF0000"></h:outputText>
      </center>
    </center>
  </f:view>
</body>
</html>
```

上述代码实现了一个简单的表单，单击“登录”按钮时会触发 JSF_Loginbean 类的 checkLogin 方法。下面介绍 Backing bean，登录页面的 Backing bean 是 JSF_Loginbean 类，代码如下：

```
package jsf;
import huizhi.Sessionbean.AdminLocal;
import javax.naming.InitialContext;
public class JSF_Loginbean{
  private String name;
  private String pwd;
  private String msg;
```

```

private String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public String getPwd() {
    return pwd;
}
public void setPwd(String pwd) {
    this.pwd = pwd;
}
public String getMsg() {
    return msg;
}
public void setMsg(String msg) {
    this.msg = msg;
}
public String checkLogin() ;
{
    try
    {
        InitialContext ctx = new InitialContext() ;
AdminLocal al = (AdminLocal)ctx.lookup("JEE_ofile/Adminbean/local");
        if(al.checkLogin(name,pwd) )
            return "login_success";
        else
        {
            this.msg = "登录失败! ";
            return null;
        }
    }
    catch(Exception e)
    {
        this.msg = "登录时系统出现异常! ";
        e.printStackTrace() ;
        return null;
    }
}
}

```

注意：checkLogin 方法通过 JNDI 找到了业务逻辑层的 AdminLocal 接口，并调用了该接口中的 checkLogin 方法完成了登录校验。登录成功返回 login_success，登录失败则返回 null。

下面介绍该部分在 faces-config.xml 文件中的配置，代码如下：

```
<managed-bean>
```

```

<managed-bean-name>Loginbean</managed-bean-name>
<managed-bean-class>jsf.JSF_Loginbean</managed-bean-class>
<managed-bean-scope>request</managed-bean-scope>
</managed-bean>
<navigation-rule>
  <from-view-id>/login.jsp</form-view-id>
  <navigation-case>
    <from-outcome>login_success</form-outcome>
    <to-view-id>/index_f.jsp</to_view_id>
    <redirect></redirect>
  </navigation-case>
</navigation-rule>

```

上述代码配置了登录页面的导航规则，当登录成功时，页面会导航到 index_f.jsp 页面，即后台首页。如图 10-2 所示为登录页面的运行效果。



图 10-2 登录页面

10.5.2 后台首页

登录成功后就进入了后台首页。在后台首页中，分页显示 ofile 表中的所有公文信息。在显示公文信息时，采用了 JSF 标签实现，其中涉及<h:dataTable>标签的使用。该视图页面的代码如下：

```

<%@page language = "java" pageEncoding = "UTF-8"%>
<%@taglib uri = "http://java.sun.com/jsf/html" prefix = "h"%>
<%@taglib uri = "http://java.sun.com/jsf/core" prefix = "f"%>
<html>
<head>
<title>公文后台管理</title>
</head>
<body>
<f:view>
  <center><h2>公文后台管理</h2></center>
  <center><a href= "addOfile_f.faces">添加公文</a></center>
  <br><center>
<h:dataTable border= "1" value= "#{OfileListbean.ofileList}" var= "OfileList"

```



```

width= "80%"first= "#{OfileListbean.beginIndex}"rows= "#{
    {OfileListbean.pageSize}">
<h:column>
    <f:facet name = "header">
        <center><h:outputText value= "公文标题"></h:outputText></center>
    </f:facet>
    <h:form>
        <h:commandLink action= "#{OfileListbean.getOfile}">
            <h:outputText value= "#{OfileList.title}"></h:outputText>
            <f:param name= "id" value= "#{OfileList.id}"></f:param>
        </h:commandLink>
    </h:form>
</h:column>
<h:column>
    <f:facet name= "header">
        <center><h:outputText value= "公文类别"></h:outputText></center>
    </f:facet ><center>
<h:outputTextvalue= "#{OfileList.category.name}"></h:outputText></center>
    </h:column>
<h:column>
    <f:facet name= "header">
        <center><h:outputText value= "时间"></h:outputText></center>
    </f:facet>
    <h:outputText value= "#{OfileList.time}"></h:outputText>
</h:column>
<h:column>
    <f:facet name= "header">
        <center><h:outputText value= "修改"></h:outputText></center>
    </f:facet>
    <h:form><center>
<h:commandLink action= "#{OfileListbean.getOfileByEdit}">
    <h:outputText value = "修改"></h:outputText>
        <f:param name= "id" value= "#{OfileList.id}"></f:param>
    </h:commandLink></center>'
    </h: form >
</h:column>
<h:column>
    <f:facet name= "header">
        <center><h:outputText value= "删除"></h:outputText></center>
    </f:facet>
    <h:form><center>
<h:commandLink action= "#{OfileListbean.delOfile}">
    <h:outputText value= "删除"></h:outputText>
        <f:param name= "id" value= "#{OfileList.id}"></f:param>
    </h:commandLink></center>

```

```

        </h:form>
    </h:column>
</h:dataTable>
</center><center>
    <h:form>
        <h:commandLink action= "#{OfileListbean.split}">
            <h:outputText value= "首页"></h:outputText>
        <f:param name= "currPage" value = "1"></f:param>
        </h:commandLink>
        <h:commandLink action = "#{OfileListbean.split}">
            <h:outputText value= "上一页"></h:outputText>
        <f:param name= "currPage"
value= "#{OfileListbean.currPage-1}"></f:param>
        </h:commandLink>
        <h:commandLink action= "#{OfileListbean.sqlit}">
            <h:outputText value= "下一页"></h:outputText>
        <f:param name= "currPage"
value= "#{OfileListbean.currPage+1}"></f:param>
        </h:commandLink>
        <h:commandLink action = "#{OfileListbean.split}">
            <h:outputText value = "尾页"></h:outputText>
            <f:param name= "currPage"
value = "#{OfileListbean.totalPage}"></f:param>
        </h:commandLink>
    </h:form>
    第<h:outputText value = "#{OfileListbean.currPage}"></h:outputText>页/共
<h:outputText value = "#{OfileListbean.totalPage}"></h:outputText>页
    </center>
</f:view>
</body>
</html>

```

上述代码通过 JSF 的 datatable 页面组件进行了分页显示。下面介绍 Backing bean，该页面的 Backing bean 是 JSF_OfileListbean 类，该类也是下面将要介绍的修改、查看和删除公文页面的 Backing bean。这里只给出该部分的相关代码：

```

package jsf;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Calendar;
import javax.faces.component.html.HtmlSelectOneMenu;
import javax.faces.context.FacesContext;
import javax.naming.*;
import huizhi.Entitybean.Category;
import huizhi Entitybean.Ofile;
import huizhi Sessionbean.CategoryLocal;
import huizhi Sessionbean.OfileLocal;

```

```

public class JSF_OfileListbean{
    private int id;
    private ArrayList<Ofile> ofileList;
    private ofile n;
    private String title;
    private int cid;
    private String content;
    private int currPage;
    private int beginIndex;
    private int pageSize=5;
    private int totalPage;
    public JSF_OfileListbean()
    {
        ofileList = new ArrayList<Ofile>() ;
        try
        {
            InitialContext ctx = new InitialContext() ;
            ofileLocal nl = (OfileLocal)ctx.lookup("JEE_ofile/Ofilebean/local");
            this.ofileList= nl.getOfileList() ;
            if(this.ofileList.size() %this.pageSize == 0)
                this.totalPage=this.ofileList.size() /this.pageSize;
            else
                this.totlaPage=this.ofileList.size() /this.pageSize+1;
        }
        catch(Exception e)
        {
            e.printStackTrace() ;
        }
        try
        {
            this.currPage=Integer.parseInt(FacesContext.
                getCurrentInstance().getExternalContext().
                    getRequestParameterMap().get("currPage"));
        }
        catch(Exception e)
        {
            this.currPage = 1;
        }
        try
        {
            this.id=Integer.parseInt(FacesContext.
                getCurrentInstance().getExternalContext().
                    getRequestParameterMap().get("id"));
        }
        catch(Exception e)

```

```

    {
        this.id=0;
    }
}
public int getId() {
    return id;
}
public void setId(int id){
    this.id = id;
}
Public ofile getN() {
    return n;
}
public void setN(Ofile n) {
    this.n = n;
}
public String getTitle() {
    return title;
}
public void setTitle(String title) {
    this.title = title;
}
public int getCid() {
    return cid;
}
public void setCid(int cid) {
    this.cid = cid;
}
public String getContent() {
    return content;
}
public void setContent(String content) {
    this.content= content;
}
public int getCurrpage() {
    return currPage;
}
public void setCurrPage(int currPage) {
    this.currPage = currPage;
}
public int getBeginIndex() {
    return beginIndex;
}
public void setBeginIndex(int beginIndex) {
    this.beginIndex = beginIndex;
}

```

```

    }
    public int getPageSize() {
        return pageSize;
    }
    public void setPageSize(int pageSize) {
        this.pageSize = pageSize;
    }
    public int getTotalPage() {
        return totalPage;
    }
    public void setTotalPage(int totalPage) {
        this.totalPage = totalPage;
    }
    public void setOfileList(ArrayList<Ofile> ofileList){
        this.ofileList =ofileList;
    }
    public ArrayList<Ofile> getOfileList()
    {
        return this.ofileList;
    }
    public String split()
    {
        if(this.currPage<1)
            this.currPage = 1;
        if(this.currPage>this.totalPage)
            this.currPage = this.totalPage;
        this.beginIndex= (this.currPage-1)*this.pageSize;
        return "split_success";
    }
}

```

Backing bean 类中包含了诸多属性，在构造方法中通过 JNDI 找到了业务逻辑层的 OfileLocal 接口，并调用该接口中的 getOfileList 方法完成了公文查询。该类的 split 方法实现了分页显示的功能。

下面介绍该部分在 faces-config.xml 文件中的配置，代码如下：

```

<managed-bean>
    <managed-bean-name>OfileListbean</ managed-bean-name>
    <managed-bean-class>jsf.JSF_OfileListbean</ managed-bean-class>
    <managed-bean-scope>request</ managed-bean-scope>
</managed-bean>
<navigation-rule>
    <from-view-id>/index_f.jsp</form-view-id>
    <navigation-case>
        <form-outcome>split_success</from-outcome>
        <to-view-id>/index_f.jsp</to-view-id>
    </navigation-case>
</navigation-rule>

```

```
</ navigation-case>
</navigation-rule>
```

后台首页的运行效果如图 10-3 所示。



图 10-3 后台首页

10.5.3 添加公文

后台首页提供了一个“添加公文”的超链接，单击该超链接后即可进入添加公文的页面。在添加公文页面，需要使用 JSF 标签创建相应的输入组件来完成公文添加。该视图页面的代码如下：

```
<%@page language = "java" pageEncoding = "UTF-8"%>
<%@taglib uri = "http://java.sun.com/jsf/html" prefix = "h"%>
<%@taglib uri = "http://java.sun.com/jsf/core"prefix = "f"%>
<html>
<head>
    <title>添加公文</title>
</head>
<body>
    <f:view>
        <center><h3>添加公文</h3><center>
        <h:form>
            <table align= "center" border = "0"/>
            <tr><td>公文标题:<h:inputText required= "true"value= "
                #{JSF_ofile.title}"
                style= "height:27px"id= "title"></h:inputText></td></tr>
            <tr><td>公文类别:<h:selectOneMenu style= "width:159px;height:26px"
                value= "#{JSF_ofile.cid}">
                <f:selectItems value = "#{JSF_new.categoryList}"/>
            </h:selectOneMenu></td></tr>
            <tr><td>公文内容:<h:inputTextarea style= "height:139px;
                width:286px"required= "true"
                value = "#{JSF_ofile.content}"id = "content"></h:inputTextarea></td></tr>
            <tr><td align= "center"><h:commandButton value= "确定"type= "submit"
                action= "#{JSF_ofile.addOfile}"></h:commandButton></td></tr>
```

```

        </table>
    </h:form>
</f:view>
</body>
</html>

```

上述代码实现了一个简单的添加公文的表单，单击“确定”按钮时会触发 JSF_AddOfilebean 类的 addOfile 方法。下面介绍 Backing bean，登录页面的 Backing bean 是 JSF_AddOfilebean 类，代码如下：

```

package jsf;

import javax.naming.*;
import huizhi.Sessionbean.CategoryLocal;
import huizhi.Sessionbean.OfileLocal;
import huizhi.Entitybean.*;
import java.util.Calendar;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import javax.faces.model.*;

public class JSF_AddOfilebean{
    private String title;
    private int cid;
    private ArrayList<SelectItem> categoryList;
    private String content;
    public JSF_AddOfilebean()
    {
        try
        {
            InitialContext ctx=new InitialContext() ;
            this.categoryList=new ArrayList<SelectItem>();
            CategoryLocal cl=(CategoryLocal)ctx.lookup("JEE_ofile/Categorybean/local");
            ArrayList<Category> temp=cl.getCategoryList();
            for(int j=0;j<temp.size();j++)
            {
                this.categoryList.add(new SelectItem(temp.get(j).getId(),
                    temp.get(j).getName()));
            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {

```

```

        this.title = title;
    }
    public int getCid() {
        return cid;
    }
    public void setCid(int cid) {
        this.cid = cid;
    }
    public String getContent() {
        return content;
    }
    public void setContent(String content){
        this.content = content;
    }
    public ArrayList<SelectItem>getCategoryList() {
        return categoryList;
    }
    public String addOfile()
    {
        try
        {
            InitialContext ctx=new InitialContext() ;
            ofileLocal nl=(OfileLocal)ctx.lookup("JEE_ofile/Ofilebean/local");
            ofile n = new ofile() ;
            Category c = new Category() ;
            c.setId(this.cid);
            n.setTitle(this.title);
            n.setCategory(c);
            n.setContent(this.content);
            n.setTime(new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").
format(Calendar.getInstance().getTime() ));
            nl.addOfile(n);
            return "success";
        }
        catch(Exception e)
        {
            e.printStackTrace() ;
            return "fail";
        }
    }
}

```

上述代码在构造方法中通过 JNDI 找到了业务逻辑层的 CategoryLocal 接口，并调用该接口的 getCategoryList 方法获得了公文类别信息。addOfile 方法通过 JNDI 找到了业务逻辑层的 OfileLocal 接口，并调用该接口的 addOfile 方法完成了公文添加，添加成功返回 success，添加失败则返回 fail。

下面介绍该部分在 faces-config.xml 文件中的配置，代码如下：

```
<managed-bean>
    <managed-bean-name>JSF_ofile</ managed-bean-name>
    <managed-bean-class>jsf.JSF_AddOfilebean</ managed-bean-class>
    <managed-bean-scope>request</ managed-bean-scope>
</managed-bean>
<navigation-rule>
    <from-view-id>/addOfile_f.jsp</from-view-id>
    <navigation-case>
        <form-outcome>success</from-outcome>
        <to-view-id>/index_f.jsp</to-view-id>
    </ navigation-case>
    <navigation-case>
        <form-outcome>fail</form-outcome>
        <to-view-id>/addOfile_fail.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```

通过上述配置代码可以看到，当公文添加成功后会导航到后台首页，当公文添加失败后会导航到 addOfile_fail.jsp 视图页面。

添加公文视图页面的运行效果如图 10-4 所示。

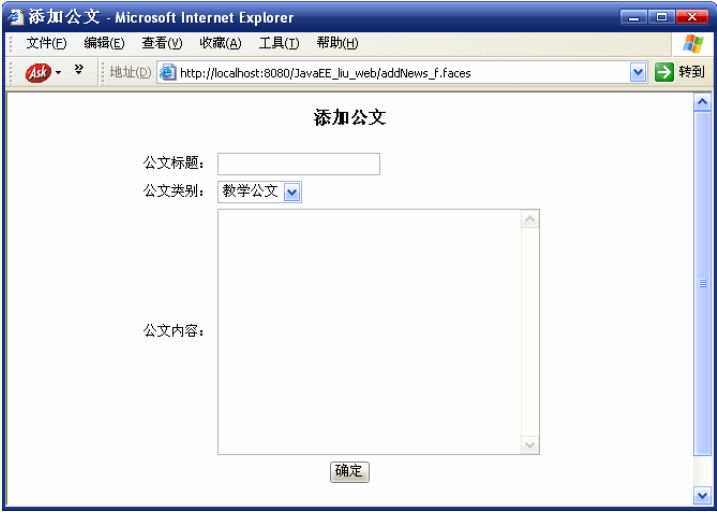


图 10-4 添加公文

10.5.4 查看公文

在后台首页中，每个公文标题都是一个超链接，单击该超链接后即可进入公文信息页面，该页面显示了公文的详细内容。实现公文显示页面，需要使用 JSF 的<h:outputText>标签，代码如下：

```
<%@page language = "java" pageEncoding = "UTF-8"%>
<%@taglib uri = "http://java.sun.com/jsf/html" prefix = "h"%>
<%@taglib uri = "http://java.sun.com/jsf/core"prefix = "f"%>
```

```

<html>
<head>
  <title>公文内容</title>
</head>
<body>
  <f:view>
    <center><h2>公文标题:
<h:outputText value= "#{OfileListbean.n.title}"></h:outputText>
      </h2></center>
<center>类别:<h:outputText value= "#{OfileListbean.n.category.name}">
      </h:outputText>&nbsp;
      &nbsp;
时间:<h:outputText value= "#{OfileListbean.n.time}"></h:outputText>
      </center>
<br><center><h:outputText value= "#{OfileListbean.n.content}">
      </h:outputText></center>
  </f:view>
</body>
</html>

```

该视图页面的 **Backing bean** 是前面介绍的 **JSF_OfileListbean** 类，下面介绍该部分的相关代码：

```

public String getOfile()
{
  try
  {
    InitialContext ctx = new InitialContext() ;
    ofileLocal nl= (OfileLocal)ctx.lookup("JEE_ofile/Ofilebean/local");
    this.n= nl.getOfileByID(this.id);
    this.title = this.n.getTitle() ;
    this.cid = this.n.getCategory().getId() ;
    this.content = this.n.getContent() ;
    return "success";
  }
  catch(Exception e)
  {
    e.printStackTrace() ;
    return null;
  }
}

```

该方法是 **JSF_OfileListbean** 类中的一个方法，通过 **JNDI** 找到了业务逻辑层的 **OfileLocal** 接口，并调用该接口的 **getOfileByID** 方法获得一个公文对象。

下面介绍该部分在 **faces-config.xml** 文件中的配置，代码如下：

```

<navigation-rule>
  <from-view-id>/index_f.jsp</form-view-id>

```

```
<navigation-case>
    <form-outcome>success</form-outcome>
    <to-view-id>/ofileInfo.jsp</to-view-id>
</navigation-case>
</navigation-rule>
```

公文信息页面的运行效果如图 10-5 所示。

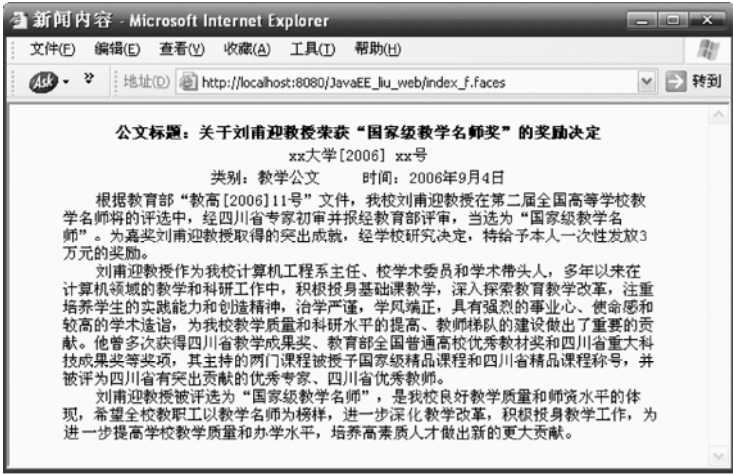


图 10-5 公文信息页面

10.5.5 修改公文

在后台首页中，每条公文都有一个“修改”超链接，单击该超链接即可进入修改公文的页面。在公文修改页面，可以修改公文标题、公文内容和公文类别。实现该页面需要使用 JSF 标签的下拉列表组件、文本区组件和文本框组件等，代码如下：

```
<%@page language = "java" pageEncoding = "UTF-8"%>
<%@taglib uri = "http://java.sun.com/jsf/html" prefix = "h"%>
<%@taglib uri = "http://java.sun.com/jsf/core" prefix = "f"%>
<html>
<head>
    <title>编辑公文</title>
</head>
<body>
<f:view>
    <center><h2>编辑公文</h2></center>
    <h:form>
        <table align= "center" border = "0"/>
        <tr><td>公文标题: <h:inputText required = "true" value=
            "#{OfileListbean.title}"
style = "height:27px" id = "title"></h:inputText></td></tr>
        <tr><td>公文类别: <h:selectOneMenu style = "width:159px;height:26px"
value= "#{OfileListbean.cid}" id = "category">
        <f:selectItems value = "#{JSF_ofile.categoryList}"/>
```

```

</h:selectOneMenu>
<h:inputHidden value = "#{OfileListbean.id}"></h:inputHidden>
</td></tr>
<tr><td>公文内容: <h:inputTextarea style= "height:139px;width:
                286px"required= "true" value=
                    "#{OfileListbean.content}" id= "content">
                        </h:inputTextarea></td></tr>
    <tr><td align = "center"><h:commandButton value = "确定"type= "submit"
action = "#{OfileListbean.editOfile}"></h:commandButton></td></tr>
</table>
</h:form>
</f:view>
</body>
</html>

```

上述代码实现了一个简单的修改公文的表单，非常简单，不再多述。该视图页面的 Backing bean 也是前面介绍的 JSF_OfileListbean 类，下面介绍该部分的相关代码：

```

public String getOfile()
{
    try
    {
        InitialContext ctx = new InitialContext() ;
        ofileLocal nl= (OfileLocal)ctx.lookup("JEE_ofile/Ofilebean/local");
        this.n= nl.getOfileByID(this.id);
        this.title = this.n.getTitle() ;
        this.cid = this.n.getCategory().getId() ;
        this.content = this.n.getContent() ;
        return "success";
    }
    catch(Exception e)
    {
        e.printStackTrace() ;
        return null;
    }
}

public String getOfileByEdit()
{
    this.getOfile() ;
    return "succ";
}

```

下面介绍该部分在 faces-config.xml 文件中的配置，代码如下：

```

<navigation-rule>
    <from-view-id>/index_f.jsp</from-view-id>
    <navigation-case>
        <form-outcome>succ</from-outcome>
        <to-view-id>/ofileInfo_edit.jsp</to-view-id>
    
```

```
</ navigation-case>
</navigation-rule>
```

修改公文页面的运行效果如图 10-6 所示。



图 10-6 修改公文

当单击“确定”按钮时就会触发 JSF_OfileListbean 类的 editOfile 方法，该方法的代码如下：

```
public String editOfile()
{
    try
    {
        InitialContext ctx = new InitialContext() ;
        ofileLocal nl= (OfileLocal)ctx.lookup("JEE_ofile/Ofilebean/local");
        ofile n = newofile() ;
        Category c = new Category() ;
        c.setId(this.cid);
        n.setTitle(this.title);
        n.setCategory(c);
        n.setContent(this.content);
        n.setTime(new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").
format(Calendar.getInstance() .getTime() ));
        nl.editOfile(n,id);
        return "edit_success";
    }
    catch(Exception e)
    {
        e.printStackTrace() ;
        return "fail";
    }
}
```

该方法通过 JNDI 找到了业务逻辑层的 OfileLocal 接口，并调用了该接口的 editOfile 方法完成了公文的更新。更新成功返回 edit_success，更新失败则返回 fail。与之对应的配置代

码如下：

```
<navigation-rule>
  <from-view-id>/ofileInfo_edit.jsp</from-view-id>
  <navigation-case><from-outcome>edit_success</from-outcome>
    <to-view-id>/index_f.jsp</to-view-id>
    <redirect/></ navigation-case>
</navigation-rule>
```

10.5.6 删除公文

在后台首页中，每条公文都有一个“删除”超链接，单击该超链接即可删除该条公文，即触发了 JSF_OfileListbean 类的 delOfile() 方法，该方法的代码如下：

```
public String delOfile()
{
    try
    {
        InitialContext ctx = new InitialContext() ;
        ofileLocal nl= (OfileLocal)ctx.lookup("JEE_ofile/Ofilebean/local");
        nl.delOfile(this.id);
        return "del_success";
    }
    catch(Exception e)
    {
        e.printStackTrace() ;
        return "fail";
    }
}
```

该方法通过 JNDI 找到了业务逻辑层的 OfileLocal 接口，并调用该接口的 delOfile 方法完成公文的删除。删除成功返回 del_success，删除失败则返回 fail。与之对应的配置代码如下：

```
<navigation-rule>
  <from-view-id>/index_f.jsp</form-view-id>
  <navigation-case>
    <from-outcome>del_success</from-outcome>
    <to-view-id>/index_f.jsp</to-view-id>
    <redirect/>
  </navigation-case>
</navigation-rule>
```

根据上述代码的导航规则配置，当公文删除成功后会返回后台首页。

习 题 10

实现本 Java EE 综合应用实例—公文管理信息系统。

附录A Java EE Web编程技术 教学大纲

(适用于高等学校计算机类专业和信息管理类专业)

一、课程的性质、任务及基本要求

Java EE Web 编程技术课程是高等学校计算机类专业和信息管理类专业理论及技术性较强的一门专业基础课。本课程系统、完整地讲述 Java EE 从规范到应用实践的主要内容。本课程的主要任务是：通过课堂教学和实习指导使学生较全面地掌握 Java EE Web 编程技术的基本概念和基本方法；初步具备使用 Java EE Web 编程技术解决实际应用课题的能力，为今后从事进行企业级 Web 应用程序设计的工作打下坚实的基础。

通过本课程的教学，应使学生达到下列基本要求：

1. 了解 Web 应用基本概念、Java EE (Java 平台企业版) 规范、Java EE 的由来与发展 (从 J2EE 到 Java EE) 和 Java EE 体系结构。
2. 了解 Java EE 的可视化集成开发平台主流技术 Eclipse、MyEclipse，其运行环境的 Web 服务器主流技术 Tomcat、应用服务器主流技术 JBoss，能用于创建、发布、部署和运行 Java EE Web 应用。
3. 了解 Java Applet (小程序) 及 JDBC 基础。
4. 掌握 Web 层编程技术 (JSP、Java Servlet、JSTL、JSF)，特别是能在 JSP 页面中使用 JSF 进行 MVC 编程。
5. 重点掌握轻型框架技术 (Hibernate、Struts2、Spring)，能用于进行中、小企业级应用程序设计。
6. 掌握分布式重型框架技术企业级 JavaBean(EJB 3.0)的应用，能用于进行大型企业级应用程序设计。
7. 会进行 Java EE 持久性数据管理。
8. 了解 Java EE 的 Web 服务 (Web Services) 技术和面向服务结构 (SOA) 新技术。
9. 了解 Ajax、Java 消息服务 (JMS) 异步技术。
10. 能进行 Java EE 综合应用程序设计。

二、教学内容

(略)

三、实践环节

见附录 B 实验指导书。

四、课时分配

序号	内 容	理论教学时数	实验时数	其他环节
1	Java EE 基础	4		
2	Java EE 的可视化集成开发平台——Eclipse 及运行环境	4	2	
3	Java Applet 及 JDBC	4	4	
4	Web 层编程技术	10	10	
5	Java EE 轻型框架技术	14	10	
6	EJB 技术	6	4	
7	Java EE 持久性数据管理	4	2	
8	Web 服务（Web Services）与 SOA 技术	6	4	
9	Ajax、Java 消息服务（JMS）异步技术	4	2	
10	Java EE 综合应用实例——公文管理信息系统	2		专周（两周）
11	合 计	58	38	专周（两周）
12	总 计	96		专周（两周）

五、说明

1. Java EE Web 编程技术是计算机类专业和信息管理类专业的主要专业课之一，在学习本课程之前的先行课程可以是 Java 程序设计、数据结构、操作系统。
2. 本课程是一门理论性和实践性都较强的课。教师在教学过程中不但要强调对理论知识的教学，更要重视学生实践能力的培养，重点放在学生应用能力的训练上，使学生能够利用所学的知识解决实际问题。
3. 在实践专周中，最好在教师的指导下，完成一个基于分布式重型框架技术企业级 JavaBean（EJB 3.0）的 Java EE 综合应用课题的设计和实现。
4. 本大纲所列学时数和实验个数均为参考数，本教学大纲主要针对软件方向，对于其他的专业方向，教师可根据实际情况对教学内容和学时数进行调整（如 64 课时）。

附录B 实验指导书

一、平时实验课（38 学时）

实验 1（2 学时）

1. 实验题目：Java EE 的可视化集成开发平台——Eclipse。
2. 目的与要求：掌握 Eclipse 平台的使用与集成开发工具的必要服务。
 - （1）下载和安装 Tomcat、JBoss、JDK 及 Eclipse SDK。
 - （2）安装 Eclipse 插件（MyEclipse 等，MyEclipse 见第 5 章）及 MySQL 数据库。
 - （3）开发 HelloWorldPlugin 插件。
 - （4）Eclipse、MyEclipse 与 Tomcat 集成。
 - （5）Eclipse、MyEclipse 与 JBoss 集成。
 - （6）使用 Eclipse、MyEclipse 平台，开发、编译、部署和运行一个 JSP 页面等。
3. 注意事项：
 - （1）安装 Eclipse SDK 时应注意硬、软件环境要求。
 - （2）成功地安装 Eclipse、MyEclipse 及 MySQL 后，了解其目录结构。
 - （3）了解 Eclipse、MyEclipse 及 MySQL 安装后的启动和关闭。
 - （4）了解 Tomcat 安装、启动、配置和首页。
 - （5）了解 JBoss 安装、启动、配置。

实验 2（4 学时）

1. 实验题目：Java Applet（小程序）及 JDBC 数据库存取技术的应用。
2. 目的与要求：掌握 Java Applet 及 JDBC 数据库存取技术的应用方法。
 - （1）了解 Java 和 Java Applet 基础。
 - （2）掌握在 HTML 和 XML 中调用 Applet 及 Applet 的事件驱动编程。
 - （3）了解 Applet 的生命周期和更复杂的 Applet。
 - （4）了解用 JDBC 访问后端数据库，掌握 JDBC 与 SQL Server 数据库联系的方法。
 - （5）根据教学要求，选择制作一个包含 JDBC 以及 Java Applet 查询技术的应用。
3. 注意事项：

（1）Java 是目前应用领域最广的语言之一，学习 Java Applet 是学习 Java 的一个必需的部分。在 WWW 网页设计中加入动画、影像、音乐等效果，使用最多的是 Java Applet 和 JavaScript。通过实验熟练掌握 Java Applet 的应用。

（2）对大多数 Applet 而言，可以在多种浏览器、平台和处理器上工作。在实验中可以根据具体情况来选择，并且 Applet 不能在浏览器对话中长驻客户机，浏览器对活动的 Applet 也有数量限制。

实验 3（10 学时）

1. 实验题目：Web 层编程技术（JSP、JSTL、Java Servlet、JSF）。
2. 目的与要求：掌握 Web 层编程技术（JSP、Java Servlet、JSTL、JSF）。

- (1) 了解 JSP 的开发环境与配置、语法概要，学会使用 JSP。
- (2) 掌握 JavaBean 的使用。
- (3) 了解 Servlet 的语法概要，用 Servlet 与 JSP、JavaBean 协同工作。
- (4) 在 JSP 中使用 JSTL。
- (5) JSP 页面中使用 JSF。展示如何在 JSP 页面中用 JSF 的 MVC 实例，如模拟航班订票系统，包括创建、发布和运行。

3. 注意事项：

(1) JSP 为创建高度动态的 Web 应用提供了一个独特的开发环境，是目前面向 Web 服务器的编程技术的热点，应通过实验熟练掌握 JSP 的应用。

(2) 用 JSP 访问后端数据库通常采用 JDBC-ODBC 桥结合 ODBC 驱动程序来完成。它可以使用多种数据库类型。在实际中可以根据实际环境，选择恰当的数据库环境，建议使用 SQL Server 数据库。

(3) 注意 Eclipse 与 Web 层应用程序的运行：其创建 Web 项目、发布 WAR 文件和运行 Web 项目等，与 1.3 节 Java EE Web 应用的编译和部署相同。

实验 4（10 学时）

1. 实验题目：轻型框架技术（Hibernate、Struts2、Spring）。

2. 目的与要求：掌握使用 Java EE 轻型框架技术（Hibernate、Struts2、Spring）编程的方法。

(1) 安装轻型框架的 MyEclipse 环境，用其开发 Java EE 应用。

(2) Struts2 的配置、Filter 及 Action。

(3) Struts2 的 OGNL 表达式和标签库应用。

(4) Hibernate 的运行及其映射、基本配置和接口。

(5) DAO 模式、Hibernate Synchronizer 插件及开发。

(6) Criteria Query、HQL 数据查询语言及 Query 接口。

(7) 体会 Hibernate 的数据关联。

(8) Hibernate 实体对象生命周期、缓存管理、事务。

(9) Spring 的 IoC、容器及基本配置。

(10) Spring 的 AOP（横切关注点、Advice、切入点、事务支持）设置。

3. 注意事项：

(1) 在进行上述实验时注意应用书上的在 Web 环境下使用 Hibernate、Spring 整合 Hibernate、开发 Struts2、Hibernate、Spring 集成程序等实例。

(2) 注意学时的合理分配。

实验 5（4 学时）

1. 实验题目：企业级 JavaBean（EJB 3.0）

2. 目的与要求：培养掌握企业级 JavaBean（EJB），用之进行大型企业级应用程序设计的能力。

(1) 了解 EJB 3.0 的特点及书上的运行实例。

(2) 用 Eclipse 开发 EJB，用独立的 Tomcat 调用 EJB。

(3) 应用 EJB 的类和接口。

(4) 应用无状态会话 Bean、有状态会话 Bean。

- (5) 应用消息驱动 Bean。
- (6) 使用实体 Bean 配置文件及 JBoss 的数据源。
- (7) 使用单表实体 Bean。

3. 注意事项:

(1) 注意掌握分布式重型框架技术企业级 JavaBean (EJB 3.0) 的应用, 能用之进行大型企业级应用程序设计。

- (2) 注意学时的合理分配。

实验 6 (2 学时)

1. 实验题目: Java EE 持久性数据管理。

2. 目的与要求: 掌握基于持久化实体管理器实现 Java EE 持久性数据管理及多表实体 Bean 的方法。

- (1) 应用 Web 层持久性。

- (2) EJB 层的持久性。

3. 注意事项:

- (1) 在进行上述实验时注意应用持久化实体管理器。

- (2) 在进行 EJB 层的持久性实验时, 注意应用书上的 roster 程序中的多对多关系。

实验 7 (4 学时)

1. 实验题目: Web 服务 (Web Services) 与 SOA 技术

2. 目的与要求: 掌握使用 Web 服务 (Web Services) 与 SOA 技术的方法。

- (1) 了解 XML: 自描述数据 (DTD 和模式语言、解析 XML)。

- (2) 使用 Web 服务技术, 利用 UDDI 注册 Web 服务。

- (3) 用 JAX-WS 构建、测试和运行 Web 服务。

- (4) 了解 SOA 的基础架构及实现。

3. 注意事项:

- (1) 注意应用书上的例子。

- (2) 特别注意体会面向服务结构 (SOA) 新技术。

实验 8 (2 学时)

1. 实验题目: Ajax、Java 消息服务 (JMS) 异步技术。

2. 目的与要求: 掌握使用 Ajax、Java 消息服务 (JMS) 异步技术的方法。

- (1) 了解 Java 消息服务 (JMS) 概念与编程模型。

- (2) 消息发布、预约、部署。

- (3) Asynchronous JavaScript+XML 的使用。

- (4) 应用 XMLHttpRequest。

- (5) 使用 Ajax 集成技术: DWR。

3. 注意事项:

- (1) 在进行上述实验时注意应用书上的例子, 如基于 Ajax 的用户注册实例等。

- (2) 注意体会异步技术。

二、实验专周 (两周)

根据学习内容, 使学生用 Eclipse +SQL Server 2000 开发、部署、运行书上基于分布式重型框架技术企业级 JavaBean (EJB 3.0) 的 Java EE 综合应用实例——公文管理信息系统。

附录C 使用日志记录

在编程时经常会使用到一些日志操作，在开发阶段需要大量的调试语句。在开发完成时，需要查找并清除。程序部署后，还经常需要一些维护调试，完成一些烦琐的日志工作。传统的方式可以使用 `System.out.println()` 向控制台输出数据，记录日志跟踪程序的运行情况。

Log4j 是用 Java 编写的优秀日志工具包，通过 Log4j 可以在不修改代码的情况下，方便灵活地控制任意力度的日志信息的开启和关闭，把日志信息输出到一个或者多个需要的地方。

一、Log4j介绍

Log4j 有 3 个主要部件：记录器（Loggers）、存放器（Appenders）和布局（Layouts）。记录器按照布局中指定的格式把日志信息写入一个或多个存放器，存放器可以是控制台、文本文件、XML 文件或 Socket 等。

Log4j 允许程序员定义多个记录器，每个记录器都有自己的名字，通过名字可以表示出记录器之间的家族关系，例如，记录器 `ab` 与记录器 `abc` 表示父子关系。每个记录器都有一个级别值，程序员可以给不同的记录器赋予不同的级别。如果某个记录器没有被赋予级别值，自动集成其父辈的值，所以记录器总有级别值。每条日志记录也有级别值，当日志记录的级别值等于或者高于记录器的级别值时，就会输出到存放器中。

在 Log4j 中，日志信息通过存放器输出到目的地，支持异步存放，一个记录器可以有多个存放器，每个记录器有一个继承机关，决定记录器是否继承其父记录器的存放器。

布局负责格式化输出的日志信息，可以让程序员以类似 C 语言的 `print()` 的格式化模板来定义格式。常见的参数如下：

`%m`: 输出代码中指定的信息。

`%p`: 输出优先级，即 `DEBUG`、`INFO`、`WARN`、`ERROR`、`FATAL`。

`%r`: 输出自应用启动到输出该 `log` 信息耗费的毫秒数。

`%c`: 输出所有的类目，通常就是所在类的全名。

`%t`: 输出产生该日志事件的线程名。

`%n`: 输出一个回车换行符。

`%d`: 输出日志时间点的日期或时间。

`%l`: 输出日志事件的存放位置，包括类目名、发生的线程，以及在代码中的行数。

二、Log4j配置

Log4j 的配置文件有两种格式：一种是 XML 文件，另一种是 Java 属性文件。这里介绍 Java 属性文件，其名字是 `log4j.properties`。在开发阶段这个文件放在 MyEclipse 项目的 `src` 目录下。在测试和发布时，MyEclipse 会把它赋值到 `WEB-INF/classes` 目录下。

Log4j 的日志级别分为 `OFF`、`FATAL`、`ERROR`、`WARN`、`INFO`、`DEBUG` 和 `ALL`。Log4j 建议只使用 4 个级别。通过这个级别，可以控制应用程序中相应级别的日志信息的输出。

日志可以输出到很多地方，Log4j 提供的 `Appender` 有以下几种：`org.apache.log4j`。

ConsoleAppender（控制台），org.apache.log4j.FileAppender（文件）和 org.apache.log4j.DailyRollingFileAppender（每天产生一个日志文件）等。

PatternLayout 布局后，可以设定参数输出格式。

其中，Log4j 提供的 layout 有以下几种：

org.apache.log4j: HTMLLayout 以 HTML 表格形式布局。

org.apache.log4j: PatternLayout 可以灵活地指定布局模式。

org.apache.log4j: SimpleLayout 包含日志信息的级别和信息字符串。

下面是一个系统的配置文件：

```
log4j: rootLogger = info,stdout,R
log4j: appender.stdout = org.apache.log4j.ConsoleAppender
log4j: appender.stdout.Target = System.out
log4j: appender.stdout.layout.ConversionPattern%d{yyyy-MM-dd HH:mm:ss:
      SSS}%c:%L%n[%-5p]:%m%n
```

其中“log4j.rootLogger=DEBUG, stdout, R”表明要显示所有优先权等于和高于 Debug 的信息。“stdout, R”表示定义了两个输出端。

后面 3 行说明 stdout 输出端其实是标准输出 Console，也就是屏幕。输出的格式是 PatternLayout。转换方式是%5p (%F: %L) -%m%n，即前五格用来显示优先权，再显示当前的文件名、当前的行数，最后是 logger.debug()、logger.info()、logger.warn() 或 logger.error() 里的信息。%n 表示回车空行。

三、加载配置文件

加载配置文件的方法很多，下面介绍最常用的在 web.xml 中加载文件的方法。在 web.xml 文件中配置一个 Servlet 加载 log4j.properties 文件，代码如下：

```
<servlet>
  <servlet-name>log4j:init</servlet>
  <servlet-class>com.apache.Jakarta.log4j.Log4jInit</servlet-class>
  <init-param>
    <param-name>log4j</param-name>
    <param-value>WEB-INF/classes/log4j.properties</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

这段配置文件是指在应用程序启动时，由 Servlet 自动加载配置文件。

四、更通用的使用方法 Commons-logging

在日志记录中除了可以使用 Log4j，还可以使用 JDK 自带的日志记录功能。它们都需要进行配置，有时候想在这两者之间进行切换，或者想简化某一种的配置，可以附加使用 Apache 开源组织开发的 Commons-logging。它为“所有的 Java 日志实现”提供一个统一的接口，并且可以简化使用和配置，避免和某一个日志系统紧密耦合。

Commons-logging 可以帮助自动选择适当的日志系统。

(1) 首先在 classpath 下寻找自己的配置文件 commons-loggings.properties，如果找到，则使用其中定义的 Log 实现类。

(2) 如果找不到 `commons-loggings.properties` 文件，则查找是否已定义系统环境变量 `org.apache.commons.logging.Log`，如果找到，使用其定义的 `Log` 实现类。

- 否则，查看 `classpath` 中是否有 `Log4j` 的包，如果发现，则自动使用 `Log4j` 作为日志实现类。
- 否则，使用 `JDK` 自身的日志实现类 (`JDK 1.4` 版本以后才有日志实现类)。
- 否则，使用 `commons-logging` 自己提供的简单的日志实现类 `SimpleLog`。

可见，`commons-logging` 总能找到一个日志实现类，并且尽可能找到一个“最合适”的日志实现类。其重要特点是：

- (1) 可以不需要配置文件。
- (2) 自动判断有没有 `Log4j` 包，有则自动使用。
- (3) 最坏的情况下也能保证提供一个日志实现 (`SimpleLog`)。

为了简化配置 `commons-loggings`，一般不使用 `commons-logging` 的配置文件，也不设置与 `commons-logging` 相关的系统环境变量，而只需要将 `log4j.jar` 包和 `log4j.properties` 放置到 `classpath` 中即可。这样就很简单地完成 `commons-logging` 与 `Log4j` 的融合。如果不想使用 `Log4j`，只需要将 `classpath` 中的 `log4j.jar` 包和配置文件删除即可。

当 `Log4j` 和 `commons-logging` 配置使用时，使用方式很简单，首先导入所需要的 `common-logging` 类，然后在自己的类中定义一个 `org.apache.commons.logging.Log` 类的私有静态变量成员：

```
private static Log log = LogFactory.getLog(UserDAO.class);
```

随后就可以使用 `org.apache.commons.logging.Log` 类的成员方法输出日志信息：

`debug()`输出“调试”级别的日志信息。

`info()`输出“信息”级别的日志信息。

`warn()`输出“警告”级别的日志信息。

`error()`输出“错误”级别的日志信息。

`fatal()`输出“致命错误”级别的日志信息。

根据不同的性质，日志信息通常划分成不同的级别，从低到高依次是：调试 (`debug`)、信息 (`info`)、警告 (`warn`)、错误 (`error`) 和致命错误 (`fatal`)。

从上面步骤可以看出，使用 `commons-logging` 的日志接口非常简单，不需要记录太多的内容。仅仅用到类 `Log`、`LogFactory`，并且两个类的方法都非常少，同时参数也非常简单。

参 考 文 献

- [1] 刘甫迎, 等. Web 编程实用技术教程. 北京: 高等教育出版社, 2009.
- [2] The Java EE 5 Tutorial.Sun, 2008.
- [3] 李芝兴, 等. Java EE Web 编程 (Eclipse 平台). 北京: 机械工业出版社, 2008.
- [4] 郑阿奇, 等. J2EE 应用实践教程. 北京: 电子工业出版社, 2009.
- [5] Kevin Mukhar, 等. Java EE 5 开发指南. 北京: 机械工业出版社, 2006.
- [6] Greg Barish, 等. J2EE Web 应用高级编程. 北京: 清华大学出版社, 2002.
- [7] 朱俊成, 等. EJB 3.0 从入门到精通. 北京: 电子工业出版社, 2009.